PRZYDATNY KOD
ZAWSZE POD RĘKĄ!

PHP i MySQL

ROZMÓWKI

Christian Wenz

Helion

Tytuł oryginału: PHP and MySQL Phrasebook

Tłumaczenie: Daniel Kaczmarek

ISBN: 978-83-246-7023-9

Authorized translation from the English language edition, entitled: PHP AND MYSQL PHRASEBOOK; ISBN 0321834631; by Christian Wenz; published by Pearson Education, Inc, publishing as Addison Wesley.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: helion@helion.pl
WWW: http://helion.pl (księgarnia internetowa, katalog książek)

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

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

Printed in Poland.

Spis treści

Spis treści

4

Praca z obiektami (i zagadnienia pokrewne)

Programowanie zorientowane obiektowo (ang. *object-oriented programming* — OOP) to jeden z najczęściej współcześnie stosowanych paradygmatów programowania. W pierwszych latach swego istnienia PHP w ogóle nie był postrzegany jako język zorientowany obiektowo. Dopiero w PHP 4 dodano pewne ograniczone mechanizmy obiektowe i znacznie je wzbogacono w PHP 5.

Niniejsza książka jest leksykonem rozmówek, a programowanie zorientowane obiektowo dość wyraźnie różni się od innych poruszanych tutaj zagadnień. Rozmówki w orientacji obiektowej zwykle są obszerne, dotyczą pewnych bardzo konkretnych przypadków albo wręcz odwrotnie — mają charakter na tyle ogólny, że wręcz można je postrzegać jako wiedzę powszechnie znaną (której poszukuje się stosunkowo rzadko).

Zastanawiałem się nad kilkoma podejściami do tego rozdziału i w końcu zdecydowałem się zaprezentować mieszankę zagadnień zorientowanych obiektowo i im pokrewnych — frazy, które ilustrują przydatność mniej trywialnych mechanizmów obiektowych dostępnych w PHP, frazy, które wyjaśniają pewne podstawowe konstrukcje obiektowe, i jeszcze kilka fraz o innym charakterze. W programowaniu obiektowym najważniejsze jest praktyczne doświadczenie, lecz sądzę, że pomimo tego informacje zawarte w tym rozdziale również od czasu do czasu okażą się pomocne.

Definiowanie klas

```
class MyClass {
}
```

W tym punkcie przedstawię zwięzły (choć zdecydowanie niepełny) opis najważniejszych mechanizmów obiektowych obsługiwanych przez PHP. Centralnym elementem orientacji obiektowej jest klasa. Klasa może zawierać stałe, właściwości i metody. Z kolei słowa kluczowe określające widoczność, takie jak `public`, `protected` i `private`, wskazują zakres dostępu do danej właściwości lub metody.

Gdy utworzona zostanie instancja klasy (służy do tego słowo kluczowe `new`), słowo kluczowe `$this` stanowi odwołanie do wywołującego obiektu. Jeżeli klasa zawiera metodę o nazwie `__construct()`, zostanie ona wywołana w momencie tworzenia instancji klasy, co widać na listingu 4.1.

Listing 4.1. Implementacja klasy (class.php)

```
<?php
  class MyClass {
    private $prop = null;

    public function setProperty($value) {
      $this->prop = $value;
    }

    public function getProperty() {
      return $this->prop;
    }

    public function __construct($value = null) {
      if ($value !== null) {
        $this->prop = $value;
      }
    }
  }

  $c = new MyClass('abc');
  echo $c->getProperty();
?>
```

UWAGA

Dostępne są trzy następujące poziomy widoczności:

- **`public`** — dostępność z każdego miejsca kodu.
- **`protected`** — dostępność jedynie z wnętrza tej samej klasy oraz z jej klas potomnych, a także z instancji tej samej klasy i klas potomnych.
- **`private`** — dostępność jedynie z wnętrza tej samej klasy oraz z instancji tej samej klasy.

Dostęp do właściwości i metod klasy można też uzyskać bez wcześniejszego tworzenia instancji tej klasy — służy do tego kontekst statyczny (i słowo kluczowe `static`). Co oczywiste, dostęp do metod statycznych nie jest możliwy za pośrednictwem `$this`. Odwołanie do bieżącej klasy zapewnia natomiast słowo `self`. Odpowiedni przykład znajduje się na listingu 4.2.

Listing 4.2. Implementacja klasy z metodą i właściwością statyczną (static.php)

```
<?php
  class MyStaticClass {
    private static $prop = 'abc';

    public static function getProperty() {
      return self::$prop;
    }
  }

  echo MyStaticClass::getProperty();
?>
```

Dziedziczenie

```
class MyDerivedClass extends MyBaseClass {
}
```

Klasa może dziedziczyć po innej klasie za pomocą słowa kluczowego `extends`. W konsekwencji nowa klasa będzie zawierać (dziedziczyć) wszystkie publiczne i chronione metody klasy bazowej. Oczywiście w klasach potomnych można pokrywać odziedziczone metody.

Dostęp do klasy przodka zapewnia słowo kluczowe `parent`. Jego składnia przypomina odwołanie statyczne, jednak takim nie jest. W wywoływanej w ten sposób metodzie można zatem używać słowa `$this`.

Kod widoczny na listingu 4.3 pokrywa konstruktor w klasie potomnej, ale także wywołuje konstruktor klasy przodka. Warto zwrócić uwagę, że sygnatury obydwóch konstruktorów różnią się od siebie. W kodzie głównym również wywoływana jest metoda klasy bazowej.

Listing 4.3. Dziedziczenie w PHP (extends.php)

```
<?php
  class MyBaseClass {
    protected $value1 = null;
    protected $value2 = null;

    protected function __construct($value = null) {
      if ($value !== null) {
        $this->value1 = $value;
      }
    }

    public function getValue1() {
      return $this->value1;
    }
  }

  class MyDerivedClass extends MyBaseClass {
    protected $value2 = null;

    public function __construct($value1 = null,
    ↪$value2 = null) {
      if ($value1 !== null) {
        parent::__construct($value1);
        if ($value2 !== null) {
          $this->value2 = $value2;
        }
```

```
      }
    }

    public function getValue2() {
      return $this->value2;
    }
  }

  $c = new MyDerivedClass('abc', 'def');
  echo '1: ', $c->getValue1(), ', 2:',
  ↪$c->getValue2();
?>
```

UWAGA

PHP nie obsługuje dziedziczenia wielokrotnego, co oznacza, że dana klasa może dziedziczyć tylko po (dokładnie) jednej innej klasie. Oczywiście klasa B może dziedziczyć po klasie A, a klasa C może dziedziczyć po klasie B, dzięki czemu klasa C będzie dziedziczyć również publiczne i chronione metody klasy A.

Korzystanie z abstrakcyjnych klas i interfejsów

```
abstract class MyAbstractBaseClass {
}
```

PHP obsługuje dwa dodatkowe mechanizmy dziedziczenia: klasy abstrakcyjne i interfejsy. Obydwie konstrukcje są do siebie podobne, lecz różnice między nimi mają charakter zasadniczy. Celem obydwóch jest wyznaczenie szablonu dla

klasy potomnej; inaczej mówiąc, klasa bazowa (albo klasy bazowe, o czym więcej powiem za chwilę) definiuje, które metody trzeba zaimplementować w nowej klasie.

Gdy używa się klasy abstrakcyjnej, dziedziczenia dokonuje się tak jak zwykle, czyli za pomocą słowa kluczowego `extends`. Nowością jest natomiast fakt, że klasa bazowa jest definiowana jako abstrakcyjna, o czym decyduje słowo `abstract`. Ponadto zbiór metod z klasy bazowej też można oznaczyć jako metody abstrakcyjne, lecz nie zawierają one wówczas żadnego kodu źródłowego:

```
protected abstract function myMethod();
```

W takim przypadku klasa potomna musi bezwzględnie implementować tak zdefiniowane metody, metody te muszą mieć identyczne sygnatury oraz takie same (lub mniej restrykcyjne) ustawienia widoczności. Na przykład jeżeli abstrakcyjna klasa bazowa definiuje abstrakcyjną metodę chronioną, wówczas klasa potomna musi implementować tę metodę jako chronioną lub publiczną. Jeżeli w klasie bazowej wskazywane są typy, w klasie potomnej muszą one być identyczne.

Jeżeli metoda abstrakcyjna zawiera dodatkowe metody, które nie są zadeklarowane jako abstrakcyjne, wówczas dziedziczy się je tak samo jak po zwykłych klasach, o ile będą one chronione lub publiczne. Ilustruje to kod źródłowy przedstawiony na listingu 4.4.

Listing 4.4. Zastosowanie klasy abstrakcyjnej (abstract.php)

```
<?php
  abstract class MyAbstractBaseClass {
    protected $value1 = null;
    protected $value2 = null;

    protected function getValue1() {
      return $this->value1;
    }

    protected function getValue2() {
      return $this->value2;
    }

    protected abstract function dumpData();
  }

  class MyAbstractClass extends
  ↪MyAbstractBaseClass {
    public function __construct($value1 = null,
    ↪$value2 = null) {
      if ($value1 !== null) {
        $this->value1 = $value1;
        if ($value2 !== null) {
          $this->value2 = $value2;
        }
      }
    }

    public function dumpData() {
      echo '1: ', $this->getValue1(), ', 2:',
      ↪$this->getValue2();
    }
  }

  $c = new MyAbstractClass('ghi', 'jkl');
  $c->dumpData();
?>
```

Podobnie jak zwykłe klasy, klasy abstrakcyjne obsługują tylko dziedziczenie po jednej klasie przodka.

„Takie samo, ale inne" jest działanie interfejsów. Interfejs wygląda jak zwykła klasa języka PHP, w której jednak zamiast słowa `class` używa się słowa `interface`. Interfejs w ogóle nie zawiera żadnej implementacji, a jedynie sygnatury funkcji. Wszystkie te funkcje muszą mieć charakter publiczny.

Definiowana klasa „dziedziczy" po interfejsie. W tym kontekście mówi się jednak zawsze, że klasa *implementuje* interfejs. Podkreśla to słowo kluczowe `implements`, które zastępuje słowo `extends`.

Pozostałe reguły są bardzo podobne jak w przypadku klas abstrakcyjnych. Wszystkie metody interfejsu muszą zostać zaimplementowane, a sygnatury funkcji muszą pozostać identyczne (włącznie z poziomem widoczności, który i tak zawsze jest publiczny). Dodatkową korzyścią jest to, że klasa może jednocześnie implementować więcej niż jeden interfejs, o ile tylko w interfejsach tych nie powtarza się nazwa żadnej z metod.

Kod widoczny na listingu 4.5 tworzy dwa interfejsy, a następnie implementuje je w jednej klasie.

Listing 4.5. Zastosowanie interfejsu (interface.php)

```
<?php
  interface MyInterface1 {
    public function getValue1();
    public function getValue2();
  }

  interface MyInterface2 {
```

```
    public function dumpData();
  }

  class MyInterfaceClass implements MyInterface1,
  ↪MyInterface2 {
    protected $value1 = null;
    protected $value2 = null;

    public function __construct($value1 = null,
    ↪$value2 = null) {
      if ($value1 !== null) {
        $this->value1 = $value1;
        if ($value2 !== null) {
          $this->value2 = $value2;
        }
      }
    }

    public function getValue1() {
      return $this->value1;
    }

    public function getValue2() {
      return $this->value2;
    }

    public function dumpData() {
      echo '1: ', $this->getValue1(), ', 2:',
      ↪$this->getValue2();
    }
  }

  $c = new MyInterfaceClass('mno', 'pqr');
  $c->dumpData();
?>
```

Zapobieganie dziedziczeniu i pokrywaniu

```
final class MyFinalBaseClass {}
public final function myMethod() {}
```

Istnieją klasy, po których nie powinno się już dziedziczyć (zwłaszcza jeśli implementują one najbardziej istotne funkcje i nie chcemy, aby inni programiści te kluczowe funkcje w jakikolwiek sposób nadpisywali). Aby temu zapobiec, w PHP udostępniono słowo kluczowe `final`. Finalną można uczynić klasę (w takim przypadku nie będzie już można w ogóle po niej dziedziczyć); można też zadeklarować metodę jako finalną (wówczas nie będzie można jej już więcej pokrywać). Dlatego kod przedstawiony na listingu 4.6 rzuci wyjątek, widoczny na rysunku 4.1.

Listing 4.6. Zapobieganie nadpisywaniu dzięki słowu kluczowemu final (final.php)

```
<?php
  class MyFinalBaseClass {
    public final function getCopyrightNotice() {
      return '&copy; by Autor Oryginalny';
    }
  }

  class MyFinalClass extends MyFinalBaseClass {
    function getCopyrightNotice() {
      return '&copy; ja!';
    }
  }
?>
```

Rysunek 4.1. Metod finalnych nie można pokrywać

Oczywiście jeśli samemu pracujesz nad implementacją klasy albo metody, deklarowanie ich jako finalnych nie wydaje się konieczne. Trzeba jednak pamiętać, że któregoś dnia zespół programistyczny może się powiększyć albo możesz zdecydować się na refaktoring własnego kodu i umieszczenie go w bibliotece udostępnianej innym programistom.

Automatyczne ładowanie

```
spl_autoload_register('myAutoloadFunction');
```

Aby możliwe było użycie klasy, musi ona zostać wcześniej zdefiniowana; w przeciwnym razie PHP rzuci błąd. Gdy jednak w momencie użycia klasy ta jeszcze nie jest dostępna, można skorzystać z drugiej szansy. Daje ją funkcja `spl_autoload_register()`, która rejestruje funkcję przeznaczoną do wywołania w sytuacji, gdy kod PHP podejmie

próbę użycia klasy dotąd niezarejestrowanej. Po wywołaniu funkcji rejestrującej kod źródłowy, który usiłował się odwołać do niezdefiniowanej klasy, zostanie wykonany ponownie. Jeżeli w tym momencie klasa będzie już dostępna, wykonanie kodu będzie przebiegać dalej w normalny sposób. W przeciwnym razie zwrócony zostanie standardowy błąd.

Standardową praktyką jest używanie z góry zdefiniowanego schematu nazewnictwa plików, które zawierają kod źródłowy klas, i umieszczanie tych plików w zdefiniowanym katalogu. W takiej konwencji klasa `MyClass` będzie się znajdować w pliku */classes/MyClass.php*, a funkcja automatycznego ładowania widoczna na listingu 4.7 załaduje klasę na żądanie.

Listing 4.7. Automatyczne ładowanie klas (autoload.php)

```
<?php
  function myAutoloadFunction($classname) {
    if (preg_match('/^[a-zA-Z_\x7f-\xff]
    ↪[a-zA-Z0-9_\x7f-\xff]*$/', $classname)) {
    // sprawdzenie poprawności nazwy klasy
      require_once "/ścieżka/do/classes/
      ↪$classname.class.php";
    }
  }

spl_autoload_register('myAutoloadFunction');
?>
```

Tak więc wywołanie w postaci:

```
$c = new UnknownClass();
```

spowoduje, że wywołana zostanie funkcja `myAutoload↪Function()`, której parametrem `$classname` będzie nazwa klasy `UnknownClass`.

Jeżeli funkcja `spl_autoload_register()` zostanie wywołana więcej niż jeden raz, zarejestrowane zostaną po kolei wszystkie funkcje automatycznego ładowania. PHP będzie więc wywoływać każdą z nich aż do momentu, gdy zdefiniowana zostanie pożądana klasa — albo gdy wszystkie te funkcje zostaną wywołane.

UWAGA

Funkcja `spl_autoload_register()` została po raz pierwszy udostępniona w PHP 5.1.2 jako element biblioteki SPL (ang. *Standard PHP Library*). We wcześniejszych wersjach języka trzeba było implementować funkcję o nazwie `__autoload()` i w niej umieszczać kod odpowiedzialny za automatyczne ładowanie. W którymś momencie rozwoju PHP obsługa funkcji `__autoload()` może zostać całkowicie wyeliminowana, dlatego najlepiej jest już teraz używać tylko funkcji `spl_autoload_register()`.

UWAGA

PHP umożliwia wywoływanie metod i właściwości klas, które nie istnieją:

- Jeżeli wywoływana jest metoda klasy i metoda ta nie istnieje, wówczas wywołana zostaje metoda `__call()` tej klasy, o ile ta metoda istnieje.
- Jeżeli odczytywana jest właściwość klasy i właściwość tej klasy nie istnieje, wówczas wykonywane są metody `__get()` i `__set()` tej klasy (przeznaczone odpowiednio do odczytywania i ustawiania właściwości), o ile metody takie istnieją.

Klonowanie obiektów

```
$b = clone $a;
```

Przyjmijmy, że zmiennej `$a` przypisano instancję klasy. Instrukcja przypisania `$b = $a` nie stworzy kopii instancji `$a`, lecz zmiennej `$b` zostanie przypisane odwołanie do `$a`. Wciąż istnieć będzie zatem tylko jedna instancja klasy.

Alternatywą jest udostępniane przez PHP słowo kluczowe `clone`, które służy do tworzenia kopii klasy:

```
$b = clone $a;
```

Czasami jednak nie chcemy tworzyć *dokładnych* kopii klas. Wyobraźmy sobie na przykład, że każda instancja klasy zawiera globalnie unikatowy identyfikator GUID. Stworzony klon instancji klasy mimo wszystko powinien posiadać identyfikator GUID, który będzie niepowtarzalny.

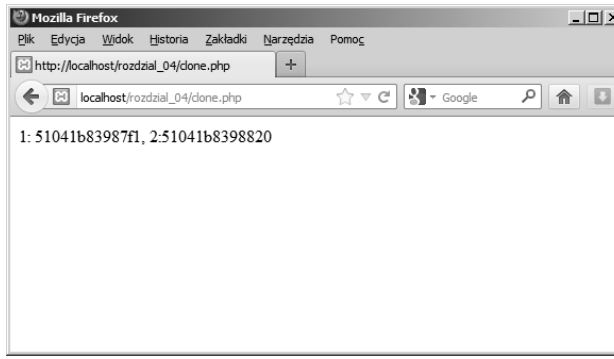
PHP pozwala na wstrzykiwanie kodu do obiektu, który został sklonowany. Jeżeli klasa posiada metodę o nazwie `__clone()`, zostanie ona wywołana na sklonowanej instancji klasy od razu po zakończeniu procesu klonowania.

Listing 4.8. Klonowanie obiektów (i zmiana ich zawartości) (clone.php)

```
<?php
  class MyCloneableClass {
    private $guid = null;

    public function __construct() {
      $this->guid = uniqid();
    }
```

```
    public function __clone() {
      $this->guid = uniqid();
    }

    public function getGuid() {
      return $this->guid;
    }
  }

  $c1 = new MyCloneableClass();
  $c2 = clone $c1;
  echo '1: ', $c1->getGuid(), ', 2:', $c2->getGuid();
?>
```

Wynik wykonania kodu z listingu 4.8 będzie podobny do widocznego na rysunku 4.2: dwie instancje tej samej klasy posiadają różne identyfikatory GUID.



Rysunek 4.2. Dwie instancje tej samej klasy posiadają różne identyfikatory GUID

Serializacja i deserializacja obiektów

```
public function __sleep() {
  $this->db->close();
  return array('guid');
}
public function __wakeup() {
  $this->connectDb();
}
```

Gdy trzeba przechowywać obiekty w celu ich wykorzystania w późniejszym czasie, trzeba te obiekty przekształcić do takiej postaci, która pozwoli na ich zapisanie na przykład w bazie danych. PHP zapewnia co najmniej kilka mechanizmów, dzięki którym instancję obiektu można przekształcić do postaci ciągu znaków (a potem stworzyć na powrót obiekt w pierwotnej postaci). Najczęściej używanym rozwiązaniem jest wykorzystanie funkcji `serialize()` i `deserialize()`, z których będę korzystał również w kolejnych rozdziałach tej książki.

Z perspektywy programowania zorientowanego obiektowo ciekawostką jest to, że proces serializacji i deserializacji można „przechwycić". Przyjmijmy na przykład, że klasa zawiera właściwość, która nie powinna (albo nie może) być serializowana, bo jest, dajmy na to, uchwytem bazy danych. Najlepiej byłoby usunąć tę właściwość przed rozpoczęciem serializacji i odtworzyć w przypadku deserializacji.

Twórcy języka PHP zdecydowali się ułatwić tego typu zadanie i zaimplementowali dwie „magiczne" metody. Jeżeli dana klasa będzie te metody implementować, zostaną one wywołane w odpowiednich okolicznościach. Metodami tymi są:

- **`__sleep()`** — musi zwracać tablicę ze wszystkimi właściwościami podlegającymi serializacji; może też czyścić serializowany obiekt.
- **`__wakeup()`** — wywołuje się ją po wykonaniu serializacji; można w niej na nowo definiować właściwości.

Kod z listingu 4.9 stanowi typową implementację mechanizmów zamykania i resetowania połączenia z bazą danych. Po wykonaniu serializacji i deserializacji identyfikator GUID pozostaje taki sam, natomiast połączenie z bazą danych jest ustanawiane na nowo.

Listing 4.9. Serializacja i deserializacja obiektów (serialize.php)

```
<?php
  class MySerializableClass {
    public $guid = null;
    public $db = null;

    public function __construct() {
      $this->guid = uniqid();
      $this->connectDb();
    }

    private function connectDb() {
      $this->db = new MySQLi('serwer', 'użytkownik',
      ↪'hasło', 'bazadanych');
    }

    public function __sleep() {
      $this->db->close();
      return array('guid');
```

```
    }

    public function __wakeup() {
      $this->connectDb();
    }
  }

  $c1 = new MySerializableClass();
  echo 'Przed: ', $c1->guid;
  $s = serialize($c1);
  $c2 = unserialize($s);
  echo '; po: ', $c2->guid;
?>
```

UWAGA

Przed deserializacją obiektu trzeba się upewnić, że dana klasa istnieje. W przeciwnym wypadku PHP użyje do stworzenia wyniku deserializacji „specjalnej" klasy o nazwie `__PHP_Incomplete_Class_Name`.

WSKAZÓWKA

PHP udostępnia całe mnóstwo „magicznych" metod i funkcji. Pełna lista takich metod i funkcji znajduje się w podręczniku online, dostępnym pod adresem *http://php.net/oop5.magic*.

Implementowanie singletonów

```
if (self::$instance == null) {
  self::$instance = new self;
}
return self::$instance;
```

Równie często stosowanym, co krytykowanym wzorcem projektowym jest wzorzec singleton. Implementacja singletonu nie jest prosta, dlatego poświęciłem mu oddzielny punkt.

Głównym celem singletonu jest to, aby niezależnie od tego, ile razy tworzy się instancję danej klasy, zawsze powstawał jeden obiekt. Takie rozwiązanie jest użyteczne na przykład w sytuacji, gdy tworzy się połączenie z bazą danych: często wystarczy w całym kodzie źródłowym używać tylko jednego połączenia z bazą.

Standardowa implementacja singletonu w PHP opiera się na kilku mechanizmach programowania zorientowanego obiektowo, których celem jest zapobieganie tworzeniu różnych instancji jednej klasy. Najczęściej konstruktor klasy staje się niedostępny przez zadeklarowanie go jako konstruktora prywatnego:

```
private function __construct() {
}
```

Jednym ze sposobów tworzenia dwóch instancji klasy jest stworzenie jednej instancji i następnie jej sklonowanie. Można temu zapobiec przez zadeklarowanie magicznej metody `__clone()` również jako metody prywatnej:

```
private function __clone() {
}
```

W takiej sytuacji nie można już stworzyć więcej niż jednej instancji tej samej klasy. Nie można nawet stworzyć *jednej* jej instancji. Dlatego trzeba zaimplementować metodę, która

stworzy instancję klasy. W kodzie klasy można wywołać jej konstruktor, ponieważ jest on zadeklarowany jako `private`. Instancja klasy będzie wówczas przechowywana we właściwości prywatnej i statycznej. Metoda odpowiedzialna za stworzenie instancji będzie zatem najpierw sprawdzać, czy instancja klasy jest już przypisana właściwości statycznej. Jeżeli tak, metoda zwróci tę instancję. W przeciwnym razie kod stworzy nową, tylko jedną instancję klasy:

```
static private $instance = null;

static public function getInstance() {
  if (self::$instance == null) {
    self::$instance = new self;
  }
  return self::$instance;
}
```

Kod z listingu 4.10 zawiera pełną implementację klasy singletonu.

Listing 4.10. Implementacja singletonu (singleton.php)

```
<?php
  class PHPSingleton{
    static private $instance = null;

    static public function getInstance() {
      if (self::$instance == null) {
        self::$instance = new self;
      }
      return self::$instance;
    }

    private function __construct() {
    }
    private function __clone() {
```

```
    }
  }
?>
```

UWAGA

Jak wspomniałem już wcześniej, wzorce singletonów mają pewne wady. Przede wszystkim w przypadku korzystania z singletonów kod staje się trudniejszy do testowania, ponieważ klasa singletonu musi być klasą globalną i nadaje stan aplikacji. Jak zwykle w takich przypadkach, ostateczne rozwiązanie zależy od charakteru rozwiązywanego problemu.

Korzystanie z przestrzeni nazw

Jednym z ważniejszych nowych mechanizmów dodanych do PHP 5.3 jest obsługa przestrzeni nazw. Przestrzenie nazw to narzędzie, które służy do grupowania powiązanych ze sobą funkcji z jednoczesnym uniknięciem konfliktów nazw. Wyobraźmy sobie, że implementowana jest funkcja o ogólnej nazwie `showInfo()`. Kolejny programista, który pracuje na tym samym projekcie, implementuje inną funkcję, ale nadaje jej tę samą nazwę. Często w takiej sytuacji nazwę funkcji poprzedza się jakimś przedrostkiem, aby uczynić ją unikatową. W efekcie powstają nieczytelne, zbyt długie identyfikatory, na przykład `Projekt_Moduł_Podmoduł_showInfo()`. Z pomocą w takich sytuacjach przychodzą przestrzenie nazw. Przy użyciu przestrzeni nazw wskazuje się kontekst, w danej przestrzeni nazw funkcja `showInfo()` jest unikatowa, a inni programiści wykorzystują inne przestrzenie nazw. W konsekwencji dzięki przestrzeniom nazw łatwiej jest enkapsulować konkretne rozwiązania.

Słowem kluczowym dla przestrzeni nazw jest słowo `namespace`. Słowo to musi być pierwszym poleceniem obecnym na stronie PHP (mogą je poprzedzać co najwyżej komentarze, które są opcjonalne); nie może przed nim występować nawet żaden znacznik języka HTML. Aby stworzyć strukturę przestrzeni nazw, można zdefiniować hierarchię, w której w roli separatora będzie się używać znaku odwrotnego ukośnika:

```
namespace Projekt\Moduł\Podmoduł;
```

Tak zdefiniowana przestrzeń nazw będzie prawidłowo funkcjonować w całym pliku. Jeżeli w pliku tym umieści się implementację metody `showInfo()` i plik ten dołączy się do innego skryptu PHP, wówczas metodę `showInfo()` będzie można wykonać następującą instrukcją:

```
Projekt\Moduł\Podmoduł\showInfo();
```

Aby w jednym pliku zdefiniować więcej niż jedną przestrzeń nazw, można wykonać kilka instrukcji `namespace`. Zalecanym podejściem jest jednak w takiej sytuacji zastosowanie nawiasów klamrowych:

```
namespace Przestrzeńnazw1 {
  //...
}

namespace Przestrzeńnazw2 {
  //...
}
```

W PHP istnieje kilka sposobów używania przestrzeni nazw. Można oczywiście podawać w pełni kwalifikowaną nazwę przestrzeni nazw, jak w przykładzie. Dozwolone jest także stosowanie nazw względnych. W przypadku przestrzeni nazw o charakterze lokalnym bardziej dogodnym podejściem jest stosowanie aliasów. Składnia `use <przestrzeńnazw>` albo `use <przestrzeńnazw> as <alias>` może znacząco skrócić czas implementacji kodu i ułatwić

korzystanie z przestrzeni nazw. Poniższy fragment kodu ilustruje sposób, w jaki importuje się przestrzeń nazw:

```
import Projekt\Moduł\Podmoduł;
Podmoduł\showInfo();
```

Jeżeli użyje się aliasu, kod może przybrać następującą postać:

```
import Projekt\Moduł\Podmoduł as Mod;
Mod\showInfo();
```

Na pierwszy rzut oka skorzystanie z wbudowanych klas PHP wewnątrz przestrzeni nazw może budzić wątpliwości, ponieważ interpreter będzie szukał wskazanych klas w bieżącej przestrzeni nazw. Wystarczy jednak poprzedzić nazwę klasy wbudowanej znakiem odwrotnego ukośnika, który pozwala PHP poszukać klasy na liście klas globalnych:

```
namespace Projekt\Moduł\Podmoduł;
$c = new \SoapClient('plik.wsdl');
```

Na temat przestrzeni nazw można by mówić jeszcze długo, jednak ze względu na formułę tej książki skupiłem się jedynie na sposobie ich działania (i to też w dużym skrócie). Zdecydowanie więcej informacji na temat przestrzeni nazw można znaleźć w podręczniku online, dostępnym pod adresem *http://php.net/namespace*.

Stosowanie cech

```
class MyTraitClass {
  use MyTrait1, MyTrait2 {
    MyTrait1::showTime as now;
    MyTrait2::showCopyright insteadof MyTrait1;
  }
}
```

Jednym z nowych rozwiązań dodanych do PHP 5.4 jest obsługa tak zwanych cech (ang. *trait*), które stanowią narzędzie ponownego wykorzystania istniejącego kodu źródłowego. Cecha jest kolekcją funkcji — swego rodzaju klasą, która nie posiada instancji. Aby skorzystać z nowego mechanizmu, trzeba najpierw stworzyć cechę, wykorzystując do tego celu słowo kluczowe `trait`.

```
trait MyTrait1 {
  public function showTime() {
    $date = new DateTime('now');
    echo $date->format('H:i:s');
  }
}
```

Następnie w kodzie klasy można takie zdefiniowane cechy ładować przy użyciu słowa kluczowego `use`. W ten sposób metody stają się dostępne wewnątrz klasy:

```
class MyTraitClass {
  use MyTrait1;
}

$c = new MyTraitClass();
$c->showTime();
```

W jednej klasie można korzystać z więcej niż jednej cechy. Trzeba jednak wówczas zwrócić szczególną uwagę na to, czy cechy te nie implementują tych samych metod. W takiej sytuacji zwrócony zostanie komunikat błędu, podobny do komunikatu rzucanego wtedy, gdy implementuje się interfejsy z metodami o takich samych nazwach. W przeciwieństwie do interfejsów cechy pozwalają jednak zapobiegać tego typu błędom. Gdy ładuje się cechę, potencjalne konflikty nazw można rozwiązywać przy użyciu operatora

instanceof. Załóżmy na przykład, że istnieją dwie cechy: MyTrait1 i MyTrait2 i obydwie te cechy zawierają definicję metody showtime(). Składnia przedstawiona poniżej zapewni, że żaden błąd się nie pojawi oraz że w kodzie użyta zostanie implementacja metody showTime() pochodząca z cechy MyTrait1 zamiast MyTrait2:

```
class MyTraitClass {
  use MyTrait1, MyTrait2{
    MyTrait1::showTime insteadof MyTrait2;
  }
}
```

Innym rozwiązaniem jest zastosowanie aliasu, użycie słowa kluczowego as, a nawet skorzystanie w cesze z metod abstrakcyjnych. W kodzie widocznym na listingu 4.11 implementowane są dwie cechy, wykorzystywany jest operator insteadof oraz alias, a wynik działania tego kodu znajduje się na rysunku 4.3.

Listing 4.11. Użycie dwóch cech (trait.php)

```
<?php
  trait MyTrait1 {
    public function showTime() {
      $date = new DateTime('now');
      echo $date->format('H:i:s');
    }

    public function showCopyright() {
      echo '&copy; Cecha 1';
    }
  }

  trait MyTrait2 {
    public function showCopyright() {
```

```
      echo '&copy; Cecha 2';
    }
  }

  class MyTraitClass {
    use MyTrait1, MyTrait2 {
      MyTrait1::showTime as now;
      MyTrait2::showCopyright insteadof MyTrait1;
    }
  }

  $c = new MyTraitClass();
  $c->now();
  echo '<hr />';
  $c->showCopyright();
?>
```

Rysunek 4.3. W przykładowym kodzie wykonana została metoda drugiej cechy

Skorowidz

C

D

E

F

G

H

I

J

K

L

O

P

R

S

Ś

T

U

W

X

Z

Ż

GOTOWE DO UŻYCIA
FRAGMENTY KODU

PHP i MySQL
ROZMÓWKI

PHP i MySQL to niewątpliwie najpopularniejszy duet do tworzenia dynamicznych witryn WWW i aplikacji internetowych. Jego popularność jest wynikiem połączenia ogromnych możliwości, przystępności oraz bezpłatnego dostępu do obu narzędzi. PHP i MySQL sprawdzą się w wielu zastosowaniach — od prostych skryptów, przez systemy do zarządzania treścią na stronach WWW, po sklepy internetowe oraz zaawansowane serwisy.

W tej książce znajdziesz dziesiątki przykładowych fragmentów kodu, które możesz od ręki wykorzystać do swoich potrzeb. Operacje na ciągach znaków i tablicach, formatowanie dat czy przetwarzanie formularzy to tylko niektóre z poruszanych tu zagadnień. Ponadto Twoją ciekawość powinien wzbudzić rozdział poświęcony programowaniu obiektowemu, korzystaniu z sesji oraz nawigowaniu po systemie plików na serwerze. W każdym z poruszanych tematów znajdziesz odwołania do repozytorium PEAR, którego zawartość pomoże Ci wiele problemów rozwiązać znacznie lepiej i szybciej. Tę książkę każdy programista PHP powinien mieć zawsze pod ręką!

Dzięki niej:

- błyskawicznie rozwiążesz typowe problemy z PHP i MySQL
- będziesz mieć pod ręką przydatne i gotowe do użycia fragmenty kodu
- sprawdzisz, jak zawartość repozytorium PEAR może Ci pomóc

DEVELOPER'S
LIBRARY

helion.pl
księgarnia
internetowa

Helion

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Cena: 39,90 zł
ISBN 978-83-246-7023-9

Nr katalogowy: 14212

Informatyka w najlepszym wydaniu