helion.pl
KSIĘGARNIA INTERNETOWA WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

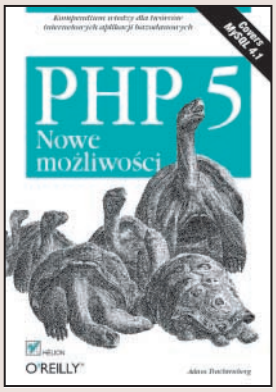
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

PHP 5. Nowe możliwości



Autor: Adam Trachtenberg
Tłumaczenie: Daniel Kaczmarek
ISBN: 83-7361-714-0
Tytuł oryginału: Upgrading to PHP 5
Format: B5, stron: 320

Przewodnik po najnowszej wersji najpopularniejszego języka do tworzenia dynamicznych witryn WWW

Książka „PHP 5. Nowe możliwości” to opis wszystkich funkcji, które dodano do najnowszej wersji języka PHP. Jest adresowana do programistów korzystających z PHP 4, którzy chcą poznać nowe narzędzia wprowadzone w wersji 5. Każdy nowy mechanizm jest przedstawiony w postaci przykładu. Książka zawiera także porównanie sposobów realizacji typowych zadań programistycznych za pomocą języków PHP 4 i PHP 5, co ułatwia migrację do nowej wersji.

- Zasady programowania obiektowego
- Współpraca z bazą danych MySQL
- Środowisko SQLite
- Mechanizmy obsługi języka XML
- Obsługa błędów za pomocą wyjątków
- Korzystanie z mechanizmów SOAP

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Przedmowa **9**

1. Wprowadzenie **15**
Dlaczego PHP 5? 16
Co nowego w PHP 5? 17
Instalowanie i konfigurowanie PHP 5 20

2. Programowanie zorientowane obiektowo **23**
Na czym polega programowanie zorientowane obiektowo? 24
Zarządzanie pamięcią 30
Klasy 33
Mechanizmy pośrednie wobec klas 38
Dziedziczenie 42
Metody magiczne 49

3. MySQL **59**
Instalacja i konfiguracja 61
Interfejs proceduralny 62
Przedtem i teraz: łączenie się z serwerem baz danych 63
Interfejs zorientowany obiektowo 66
Przedtem i teraz: wykonywanie zapytań i pozyskiwanie danych przy użyciu instrukcji przygotowywanych 67
Przedtem i teraz: podzapytania 75
Transakcje 81
Przedtem i teraz: wykonywanie zapytań wielokrotnych 84
Zabezpieczanie połączeń przy użyciu protokołu SSL 88
Przenoszenie kodu i migrowanie baz danych 91

4. SQLite **101**
Podstawy SQLite 102
Zmieniające się typy wyników SQLite 105
Interfejs zorientowany obiektowo 106
Indeksy, obsługa błędów i tabele przechowywane w pamięci 108
Transakcje 112
Funkcje zdefiniowane przez użytkownika 114

5. XML 121
Rozszerzenia XML w PHP 5 121
Instalacja mechanizmów obsługi XML i XSLT 124
DOM 125
SimpleXML 132
Przekształcenia między obiektami SimpleXML i DOM 133
Przedtem i teraz: wczytywanie dokumentów XML do drzewa 134
Przedtem i teraz: przeszukiwanie dokumentów XML przy użyciu XPath 139
Wczytywanie kodu XML jako zdarzeń przy użyciu SAX 146
Przedtem i teraz: tworzenie nowych dokumentów XML 146
Przedtem i teraz: przekształcanie dokumentów XML przy użyciu XSLT 150
Weryfikacja zgodności ze schematem 155

6. Iteratory i standardowa biblioteka PHP SPL 157
Przedtem i teraz: używanie iteratorów 159
Implementacja interfejsu iteratora 162
Iterator wyników zapytań do bazy MySQL 164
Łańcuchowe łączenie iteratorów 167
Iterator SimpleXML 169
Przedtem i teraz: rekurencyjna iteracja po katalogu 170
Implementacja interfejsu RecursiveIterator 173
Iterowanie przez tablice i właściwości obiektów 175
Zmiana przebiegu iteracji przez klasy 177
Iteratory oraz klasy i interfejsy biblioteki SPL 180

7. Obsługa błędów i debugowanie 183
Przedtem i teraz: obsługa błędów 183
Korzyści z używania wyjątków 186
Wyjątki systemowe 187
Klasa Exception 188
Wyjątki generowane przez użytkownika 189
Definiowanie własnego uchwytu wyjątków 194
Przetwarzanie błędów we własnym uchwycie błędów 195
Funkcje debugujące 196

8. Strumienie, nakładki i filtry 201
Używanie API strumieni 202
Ogólne informacje o nakładkach 204
Szczegółowe informacje na temat nakładek 206
Tworzenie nakładek 214
Filtrowanie strumieni 223
Tworzenie filtrów 227

9. Inne rozszerzenia 231
SOAP 231
Tidy 239
Klasy Reflection 243

10. PHP w akcji 251
Definicja schematu bazy danych 252
Klasa Person 253
Klasa addressBook 257
Klasa Template 262
Złożenie aplikacji w całość 266
Nakładki i kierunki dalszego rozwoju 270

A Wprowadzenie do języka XML 273

B Pozostałe nowe mechanizmy oraz pomniejsze zmiany 283

C Instalowanie PHP 5 obok PHP 4 291

Skorowidz 299

ROZDZIAŁ 2.

Programowanie zorientowane obiektowo

Niniejszy rozdział zawiera podstawowe informacje na temat programowania zorientowanego obiektowo (ang. *object-oriented programming* — OOP) oraz przedstawia wszystkie mechanizmy zorientowane obiektowo (ang. *object-oriented* — OO) dostępne w PHP 5. Nie zakładano w nim żadnej znajomości technik OOP, dlatego nic nie stoi na przeszkodzie, by przeczytał go początkujący programista.

Rozdział ten zawiera jednak również wiele cennych informacji dla programistów PHP 4. Oprócz udostępnienia wielu mechanizmów OO, w PHP 5 zmieniono również podstawowe mechanizmy OO dostępne dotychczas w PHP 4. Jeśli w programach napisanych w PHP 4 nie zostaną dokonane odpowiednie uaktualnienia, ich uruchomienie w PHP 5 może skutkować otrzymaniem nieoczekiwanych wyników i błędów.

Nowe mechanizmy zawarte w PHP 5 pozwalają ponadto implementować najlepsze rozwiązania OOP, które w PHP 4 były po prostu niedostępne. W rozdziale tym zostanie pokazane, jak i dlaczego powinno się zmodyfikować istniejący kod, by wykorzystać wszystkie zalety PHP 5.

Pierwsze wersje języka PHP były wersjami wybitnie proceduralnymi: można było definiować funkcje, ale nie obiekty. W PHP 3 wprowadzono obiekty w postaci szczątkowej, napisane tak naprawdę jako ekstra-dodatek. W 1997 roku nikt nie spodziewał się takiego przyrostu liczby programistów PHP, nikt też nie planował pisania w tym języku rozbudowanych programów. Z tego względu istniejące ograniczenia nie stanowiły wówczas wielkiego problemu.

Przez kolejne lata PHP wzbogacał się o kolejne mechanizmy obiektowe. Zespół programistyczny nigdy jednak nie pokusił się o przepisanie kodu jądra PHP sterującego mechanizmami obiektowymi, by odpowiednio obsługiwać obiekty i klasy. W efekcie pomimo tego, że PHP 4 odznaczał się znacznie lepszą ogólną wydajnością, pisanie w nim złożonych programów OO wciąż było trudne, a niekiedy nawet niemożliwe.

W PHP 5 problemy takie odeszły w zapomnienie dzięki modułowi Zend Engine 2. Jego pierwsza wersja została napisana dla PHP 4 i obsługiwała funkcjonalności jądra języka, takie jak wyznaczanie dozwolonych typów obiektów, a także definiowała składnię PHP.

Zend Engine 2, stanowiący jądro PHP 5, umożliwia wykorzystanie bardziej zaawansowanych mechanizmów zorientowanych obiektowo, zachowując jednocześnie daleko idącą wsteczną zgodność z dotychczas napisanymi milionami skryptów PHP.

Jeśli czytelnik programował obiektowo jedynie w PHP, może na początku przeżyć zaskoczenie. Niektóre nowe mechanizmy ułatwiają realizację części zadań, lecz wiele z tych rozwiązań nie pozwala tworzyć niczego nowego. W wielu przypadkach wręcz **ograniczają** one dostępne możliwości.

Paradoksalnie jednak ograniczenia te pozwalają tak naprawdę na szybkie pisanie bezpiecznego kodu, ponieważ ułatwiają ponowne wykorzystanie kodu oraz enkapsulację danych. Kluczowe koncepcje programowania zorientowanego obiektowo zostaną objaśnione w dalszych punktach rozdziału.

Na czym polega programowanie zorientowane obiektowo?

Programowanie zorientowane obiektowo stanowi sposób grupowania funkcji i danych w jeden blok. Blok ten określany jest mianem **obiektu**.

Wielu użytkowników preferuje OOP, ponieważ pozwala ono na tak zwaną **enkapsulację** danych. Zawsze w trakcie pisania kodu okazuje się, że niektóre jego części — sposób przechowywania danych, zakres parametrów pobieranych przez funkcję, sposób organizacji bazy danych — nie działają tak dobrze, jak powinny. Okazują się one zbyt wolne, zbyt niewygodne albo uniemożliwiają rozszerzanie ich o nowe możliwości, a zatem trzeba je wyczyścić.

Poprawianie kodu jest chwalebne, ale tylko do czasu, gdy przypadkiem dojdzie do uszkodzenia innych części systemu biorących udział w poprawianym procesie. Jeżeli program zostanie zaprojektowany na wysokim poziomie enkapsulacji, używane przez niego struktury danych i tabele bazodanowe nie są wykorzystywane bezpośrednio. Zamiast tego definiuje się zestaw funkcji, które pełnią rolę pośredników w procesie przepływu wszystkich wywołań.

Załóżmy na przykład, że istnieje tabela bazy danych przechowująca nazwiska i adresy poczty elektronicznej. Program, w którym enkapsulacja jest słaba, odwołuje się bezpośrednio do tabeli za każdym razem, gdy konieczne jest odczytanie adresu pocztowego danej osoby:

```
$name   = 'Rasmus Lerdorf';
$db     = mysql_connect();
$result = mysql_query("SELECT email FROM users
                       WHERE name LIKE '$name'", $db);
$row    = mysql_fetch_assoc($result);
$email  = $row['email'];
```

W programie o lepszej enkapsulacji użytoby natomiast funkcji:

```
function getEmail($name) {
   $db = mysql_connect();
   $result = mysql_query("SELECT email FROM users
                          WHERE name LIKE '$name'", $db);
   $row = mysql_fetch_assoc($result);
   $email  = $row['email'];
   return $email;
}

$email = getEmail('Rasmus Lerdorf');
```

Użycie funkcji `getEmail()` wiąże się z wieloma zaletami, między innymi ze zmniejszeniem ilości kodu, jaki trzeba napisać w celu pobrania adresu pocztowego. Poza tym pozwala to na bezpieczne wprowadzenie zmian w konstrukcji bazy danych, ponieważ trzeba wówczas zmienić tylko jedno zapytanie występujące w funkcji `getEmail()`, a nie przeszukiwać cały kod w każdym z plików pod kątem zapytania `SELECT` wyszukującego dane w tabeli `users`.

Napisanie jedynie przy użyciu funkcji programu charakteryzującego się dobrą enkapsulacją jest trudne, ponieważ jedynym sposobem zasygnalizowania programiście, by nie dotykał jakiegoś fragmentu kodu jest zawarcie odpowiedniej informacji w komentarzach lub zastosowanie konwencji programistycznych.

Obiekty pozwalają na odgrodzenie wewnętrznych mechanizmów implementacyjnych od dostępu z zewnątrz. Dzięki temu inni programiści nie mogą odwoływać się do kodu, który w przyszłości może się zmienić, lecz są zmuszeni korzystać jedynie z udostępnionych funkcji, aby odczytać dane. Tego typu funkcje to tak zwane **funkcje dostępu** (ang. *accessors*), ponieważ umożliwiają uzyskanie dostępu do informacji mających w innych okolicznościach status chronionych. W przypadku zmian w kodzie wystarczy tylko uaktualnić funkcje dostępu tak, by działały jak dotychczas. Wówczas cała reszta kodu dalej będzie działać prawidłowo.

Więcej informacji na temat enkapsulacji zostanie zamieszczonych później, najpierw jednak przejdziemy do wprowadzenia do używania obiektów w PHP 5.

Używanie obiektów

Zazwyczaj obiekty reprezentują rzeczywiste lub namacalne jednostki „świata rzeczywistego", na przykład osobę. Oto zapisana w PHP jedna z możliwych wersji obiektu `Person` reprezentującego osobę:

```
$rasmus = new Person;
$rasmus->setName('Rasmus Lerdorf');
print $rasmus->getName();
Rasmus Lerdorf
```

W pierwszym wierszu zmiennej `$rasmus` przypisywana jest wartość. Jest ona obiektem typu `Person`. `Person` to wcześniej zdefiniowana struktura zawierająca kod opisujący sposób, w jaki „obiekt osoba" powinien się zachowywać. Struktura taka nosi nazwę **klasa**.

Różnica między obiektem i klasą polega na tym, że obiekt jest zmienną, którą można manipulować. Można ją przekazywać do funkcji, usuwać, kopiować i tak dalej. Zmienna ta przechowuje określony zestaw danych.

Klasa jest zaś szablonem definiującym sposób, w jaki można używać obiektu, oraz dane, jakie może on przechowywać.

Klasę przekształca się w obiekt przy użyciu słowa kluczowego `new`:

```
$rasmus = new Person;
```

Polecenie to nakazuje PHP odszukanie klasy o nazwie `Person`, utworzenie jej nowej kopii i przypisanie tej kopii do zmiennej `$rasmus`. Proces ten to tak zwane **tworzenie egzemplarza** obiektu lub tworzenie nowej kopii klasy.

Na obecnym etapie nie trzeba się martwić o rzeczywistą składnię służącą do definiowania obiektu `Person`. Niepotrzebna jest również znajomość sposobu, w jaki `Person` przechowuje dane. Informacja ta podlega enkapsulacji i trzeba się obyć bez niej (a to bardzo dobrze!).

Trzeba natomiast wiedzieć, że `Person` pozwala na wywoływanie czegoś, co przypomina funkcję o nazwie `setName()`:

```
$rasmus->setName('Rasmus Lerdorf');
```

W momencie definiowania klasy można wskazać funkcje, które będą do niej należały. Aby wywołać funkcje obiektu, trzeba po tym obiekcie wpisać symbol strzałki (`->`), a następnie wskazać nazwę funkcji. Nakazuje to PHP, by wywołał funkcję `setName()` na tej konkretnej kopii klasy.

Właściwym określeniem na `setName()` nie jest „funkcja". Tak naprawdę jest to **metoda** lub **metoda obiektu**. W pełnym zdaniu pojęć tych należy używać w następujący sposób: „Wywołałem metodę `setName()` na obiekcie" albo „Musisz wywołać metodę `setName()` obiektu".

Metoda `setName()` przypisuje atrybutowi `name` wartość zmiennej `$rasmus`. W powyższym przykładzie wartością tą jest `Rasmus Lerdorf`.

Wartość tę można odczytać, wywołując metodę `getName()`:

```
print $rasmus->getName();
Rasmus Lerdorf
```

Metoda `getName()` wyszukuje wartość zapisaną w wyniku wcześniejszego wywołania metody `setName()` i ją zwraca. Ze względu na enkapsulację nie wiadomo, w jaki sposób obiekt `Person` przechowuje dane — zresztą takie szczegóły nie są do niczego potrzebne.

Szczegółowe informacje na temat sposobu tworzenia klas zostaną przedstawione później, poniżej natomiast przedstawiono elementy prostej klasy. Klasa `Person` mogłaby na przykład wyglądać następująco:

```
class Person {
   setName($name) {
      $this->name = $name;
   }

   getName() {
      return $this->name;
   }
}
```

Klasę i jej nazwę definiuje się w taki sam sposób, jak definiuje się funkcję i jej nazwę. Jedyne różnice polegają na tym, że zamiast słowa kluczowego `function` trzeba użyć słowa `class`, a po nazwie klasy nie umieszcza się znaków nawiasu (`()`).

Wewnątrz klasy deklaruje się jej metody w taki sam sposób, w jaki deklaruje się zwykłe funkcje:

```
function setName($name) {
   $this->name = $name;
}

function getName() {
   return $this->name;
}
```

Powyższe dwie metody zapisują i zwracają nazwisko `name` przy użyciu specjalnej zmiennej klasy o nazwie `$this`. Dzięki temu metoda `$rasmus->getName()` potrafi zapamiętać i zwrócić wartość przekazaną do niej przez `$rasmus->setName('Rasmus Lerdorf')`.

Na razie to wszystko na temat tworzenia klas. Pora wrócić do używania klas i obiektów.

W PHP 5 można wywołać metodę na obiekcie zwróconym przez funkcję:

```
function getRasmus() {
   $rasmus = new Person;
   $rasmus->setName('Rasmus Lerdorf');
   return $rasmus;
}

print getRasmus()->getName();
Rasmus Lerdorf
```

W PHP 4 nie byłoby to możliwe. Zamiast tego, jako krok pośredni, należałoby zapisywać obiekt w zmiennej tymczasowej:

```
function getRasmus() {
   $rasmus = new Person;
   $rasmus->setName('Rasmus Lerdorf');
   return $rasmus;
}

$rasmus = getRasmus();
print $rasmus->getName();
```

Wywołanie metody `setName()` na różnych obiektach doprowadzi do uruchamiania tej metody na różnych zestawach danych. Każda jej kopia będzie działać niezależnie od wszystkich pozostałych kopii, nawet jeśli będą one pochodzić z tej samej klasy.

```
$rasmus = new Person;
$zeev   = new Person;

$rasmus->setName('Rasmus Lerdorf');
$zeev->setName('Zeev Suraski');

print $rasmus->getName();
print $zeev->getName();
Rasmus Lerdorf
Zeev Suraski
```

Przykład ten tworzy dwie kopie klasy `Person`: `$rasmus` i `$zeev`. Obiekty te są od siebie niezależne, zatem wywołanie `$zeev->setName('Zeev Suraski');` nie zmieni wyniku wcześniejszego wywołania `$rasmus->setName('Rasmus Lerdorf');`.

Obiekty, oprócz metod, mogą posiadać także właściwości. Właściwość jest dla obiektu tym, czym element jest dla tablicy. Odwołuje się do niej za pośrednictwem jej nazwy, a może ona przechowywać dane różnych typów: łańcuchy znaków, tablice, a nawet inne obiekty.

Składnia instrukcji uzyskującej dostęp do właściwości przypomina składnię instrukcji uzyskującej dostęp do metody, tyle tylko, że po nazwie właściwości nie wpisuje się nawiasów:

```
$rasmus = new Person;
$rasmus->name = 'Rasmus Lerdorf';
print $rasmus->name;
Rasmus Lerdorf
```

Kod ten przypisuje łańcuch znaków `Rasmus Lerdorf` do właściwości `name` obiektu `$rasmus`. Następnie łańcuch ten jest odczytywany i wyświetlany. Obiekt, który zawiera jedynie właściwości i nie zawiera metod, jest mniej lub bardziej wyszukaną tablicą.

Automatyczne ładowanie

Jeśli podjęta zostanie próba utworzenia egzemplarza klasy, która nie została zdefiniowana, PHP 4 zwróci błąd krytyczny, ponieważ nie zdoła zlokalizować szukanej struktury. PHP 5 rozwiązuje ten problem, ładując brakujący kod w locie przy użyciu nowego mechanizm automatycznego ładowania.

Częste używanie klas wymaga, by zdefiniować je wszystkie w jednym pliku lub na początku każdego skryptu używającego klasy umieścić odpowiednią instrukcję `include`. Ponieważ PHP 5 wywołuje metodę `__autoload()` za każdym razem, gdy tworzony jest egzemplarz klasy niezdefiniowanej, można wykorzystać instrukcję `include` i niewielkim nakładem pracy załadować nią wszystkie klasy używane w skrypcie:

```
function __autoload($nazwa_klasy) {
   include "$nazwa_klasy.php";
}

$person = new Person;
```

Funkcja `__autoload()` pobiera jako jedyny parametr nazwę klasy. W powyższym przykładzie do nazwy tej doklejane jest rozszerzenie *.php*, po czym następuje próba dołączenia pliku o nazwie wyznaczanej przez `$nazwa_klasy`. Zatem w chwili tworzenia nowego egzemplarza klasy `Person` w lokalizacjach wskazanych w opcji `include_path` wyszukiwany jest plik *Person.php*.

Jeśli używana będzie konwencja nazewnictwa stosowana w PEAR, według której między poszczególnymi słowami wpisuje się znak podkreślenia odzwierciedlający hierarchię plików, można użyć kodu z listingu 2.1.

Listing 2.1. Automatyczne ładowanie klas przy użyciu konwencji nazewnictwa PEAR

```
function __autoload($package_name) {
// wydzielenie fragmentów oddzielonych podkreśleniami
$folders = split('_', $package_name);
// połączenie fragmentów w sposób oddający strukturę katalogów
// dzięki użyciu stałej DIRECTORY_SEPARATOR funkcja działa na wszystkich platformach
$path = join(DIRECTORY_SEPARATOR, $folders);
// doklejenie rozszerzenia
$path .= '.php';
include $path;
}
```

Po wpisaniu kodu z listingu 2.1 można wykonać następującą instrukcję:

```
$person = new Animals_Person;
```

Jeśli klasa nie została nigdzie zdefiniowana, `Animals_Person` zostanie przekazana do funkcji `__autoload()`. Wydzieli ona elementy nazwy klasy rozdzielone znakiem podkreślenia (_) i połączy je z powrotem, tym razem oddzielając je od siebie wartością stałej `DIRECTORY_SEPARATOR`. W efekcie, w systemach z rodziny Unix utworzony zostanie łańcuch `Animals/Person` (w systemie Windows natomiast łańcuch będzie miał wartość `Animals\Person`).

W kolejnym kroku doklejane jest rozszerzenie *.php* i dołączany jest plik *Animals/Person.php*.

Użycie `__autoload()` nieco wydłuża czas przetwarzania spowodowany koniecznością dodania klasy, lecz funkcja ta jest dla każdej klasy wywołana tylko raz. Występowanie wielu kopii tej samej klasy nie powoduje wielokrotnego wywoływania funkcji `__autoload()`.

Enkapsulacja danych

Używanie właściwości zamiast metod dostępu zmniejsza wprawdzie wymagany nakład pracy, lecz nie jest najlepszym rozwiązaniem, ponieważ ogranicza enkapsulację. Odczytywanie i zapisywanie wartości bezpośrednio w `name` zamiast wywoływania `setName()` i `getName()` narusza warstwę abstrakcji, która zapobiega błędnemu działaniu kodu po wprowadzeniu zmian koncepcyjnych. Jest to niepodważalna zaleta programowania zorientowanego obiektowo, dlatego nie powinno się używać właściwości zamiast metod dostępu.

PHP 5 pozwala wymuszać rozróżnianie elementów, które powinny i nie powinny być dostępne bezpośrednio. Wszystkie metody i właściwości przedstawione dotychczas były metodami i właściwościami **publicznymi**. Oznacza to, że każdy może je wywoływać i edytować.

W PHP 4 wszystkie właściwości i metody są publiczne. W PHP 5 natomiast można używać etykiety `private`, aby zawężać dostęp jedynie do metod zdefiniowanych wewnątrz klasy. Jeśli etykietą tą poprzedzi się metodę lub właściwość, będą one miały charakter **prywatny**. Oznaczenie jakiegoś elementu jako prywatnego będzie oznaczać, że w przyszłości może on ulec zmianom i inni użytkownicy nie powinni się do niego odwoływać, ponieważ w przeciwnym razie pogwałcą zasady enkapsulacji.

Zasada ta jest czymś więcej niż tylko przyjętą konwencją. PHP 5 tak naprawdę uniemożliwia użytkownikom wywoływanie metody prywatnej lub odczytywanie prywatnej właściwości spoza klasy. Zatem patrząc z zewnątrz, tego typu metody i właściwości mogłyby równie dobrze w ogóle nie istnieć, ponieważ uzyskanie dostępu do nich i tak jest niemożliwe. Więcej informacji na temat kontroli dostępu zostanie przedstawionych w punkcie „Ograniczenia dostępu” w dalszej części rozdziału.

Konstruktory i destruktory

W PHP 5 obiekty potrafią również wywoływać konstruktory i destruktory. **Konstruktor** to metoda wywoływana automatycznie w momencie, gdy tworzony jest egzemplarz obiektu. Zależnie od tego, w jaki sposób konstruktor zostanie zaimplementowany, można przekazywać do niego argumenty.

Na przykład konstruktor klasy reprezentującej bazę danych może przyjmować jako argument adres bazy danych, z którą należy nawiązać połączenie, a także nazwę użytkownika i hasło wymagane do uwierzytelnienia się:

```
$db = new Database('db.przyklad.com', 'web', 'jsd6w@2d');
```

Instrukcja ta spowoduje utworzenie nowej kopii klasy `Database` i przekazanie do jej konstruktora trzech danych. Konstruktor klasy wykorzysta te dane do utworzenia połączenia z bazą danych, po czym zapisze otrzymany w wyniku uchwyt we właściwości prywatnej.

W PHP 4 istnieją konstruktory obiektów, lecz **destruktory** obiektów są nowością wprowadzoną w PHP 5. Destruktory przypominają konstruktory, z tą różnicą, że są one wywoływane w momencie usuwania obiektów. Nawet jeśli programista nie usunie obiektu samodzielnie wywołując metodę `unset()`, PHP 5 i tak wywoła destruktor gdy tylko zauważy, że obiekt nie będzie więcej używany. Sytuacja taka może zajść w chwili dojścia do końca skryptu, ale może nastąpić również znacznie wcześniej.

Destruktory służą do czyszczenia pamięci po obiekcie. Na przykład destruktor klasy `Database` kończyłby połączenie z bazą danych i uwalniał przydzieloną mu pamięć. W odróżnieniu od konstruktorów, do destruktorów nie można przekazywać informacji, ponieważ nie można mieć całkowitej pewności, kiedy zostaną one uruchomione.

Zarządzanie pamięcią

W PHP 4 kopiowanie zmiennej lub przekazywanie jej do funkcji nie oznacza, że przekazywana jest oryginalna zmienna. Przekazywana jest jedynie kopia danych przekazywanych w zmiennej. Jest to tak zwane przekazywanie przez wartość, ponieważ kopiowana jest wartość zmiennej i tworzony jest duplikat.

W efekcie ta nowa zmienna jest całkowicie oddzielona od zmiennej oryginalnej. Zmodyfikowanie jednej nie wpłynie w żaden sposób na drugą, podobnie jak w poprzednim przykładzie wywołanie `$zeev->setName()` nie miało wpływu na wartość `$rasmus`.

Odwołania do obiektów

W PHP 5 obiekty zachowują się inaczej niż pozostałe zmienne. Nie można przekazywać ich przez wartość, jak w przypadku wartości skalarnych albo tablic, lecz trzeba robić to przez odwołanie. **Odwołanie** albo **odwołanie do obiektu** jest wskaźnikiem do zmiennej. Zatem wszelkie zmiany dokonane na przekazanym obiekcie będą tak naprawdę dokonywane na obiekcie oryginalnym.

Oto przykład:

```
$rasmus = new Person;
$rasmus->setName('Rasmus Lerdorf');

$zeev = $rasmus;
$zeev->setName('Zeev Suraski');

print $rasmus->getName();
Zeev Suraski
```

W tym przypadku zmiana w `$zeev` spowodowała zmianę w `$rasmus`!

W PHP 4 sytuacja taka nie miałaby miejsca. PHP 4 wyświetli wartość Rasmus Lerdorf, ponieważ przypisanie `$zeev = $rasmus` spowoduje, że PHP utworzy kopię obiektu oryginalnego i przypisze ją do zmiennej `$zeev`.

W PHP 5 natomiast polecenie to spowoduje, że zmiennej `$zeev` przypisane zostanie odwołanie do `$rasmus`. Jakiekolwiek zmiany w `$zeev` będą tak naprawdę wykonywane w `$rasmus`.

Podobnie rzecz będzie się miała wówczas, gdy obiekty będą przekazywane do funkcji:

```
function editName($person, $name) {
   $person->setName($name);
}

$rasmus = new Person;
$rasmus->setName('Rasmus Lerdorf');

editName($rasmus, 'Zeev Suraski');
print $rasmus->getName();
Zeev Suraski
```

Zwykle zmiany dokonane wewnątrz `editName()` nie spowodują zmiany wartości zmiennych na zewnątrz funkcji, a aby zmienić obiekt oryginalny, trzeba zwrócić zmodyfikowaną zmienną instrukcją `return`. Tak właśnie postępuje się w PHP 4.

W PHP 5 obiekty są przekazywane przez odwołanie, dlatego dokonanie w nich zmian wewnątrz funkcji lub metody doprowadzi do zmiany obiektu oryginalnego. Nie ma potrzeby, by przekazywać je jawnie przez odwołanie albo zwracać ich zmodyfikowaną kopię. Czynność taka jest znana również jako **przekazywanie uchwytu do obiektu**, ponieważ „uchwyt" jest synonimem odwołania albo wskaźnika.

Inne rodzaje zmiennych, jak łańcuch znaków czy tablice, wciąż domyślnie są przekazywane przez wartość, chyba że w prototypie funkcji zostanie określony inny sposób, to znaczy przed nazwą zmiennej znajdzie się znak ampersanda (`&`).

Ta zmiana wprowadzona w PHP 5 znacznie ułatwia używanie obiektów, ponieważ obiekty o wiele częściej przekazuje się przez odwołanie niż przez wartość. Jeśli dane podlegają enkapsulacji wewnątrz obiektów, często przekazuje się jedną lub dwie kopie do metody i dopiero w jej wnętrzu dokonuje się zmian obiektów.

Gdyby nie to rozwiązanie, przeniesienie poczynionych zmian z powrotem do oryginalnych kopii obiektów wymagałoby umieszczenia ampersanda w każdym miejscu, w którym PHP powinien przekazywać obiekt przez odwołanie. Jeżeli jednak pominięty zostanie choć jeden ampersand, w kodzie pojawi się trudny do zidentyfikowania błąd.

Aby skopiować wewnątrz obiektu dane, a nie tylko odwołanie do niego, zamiast wykonywać bezpośrednie przypisanie realizowane przy użyciu znaku równości (=) trzeba użyć operatora `clone`:

```
$rasmus = new Person;
$rasmus->setName('Rasmus Lerdorf');

$zeev = clone $rasmus;
$zeev->setName('Zeev Suraski');

print $rasmus->getName();
print $zeev->getName();
Rasmus Lerdorf
Zeev Suraski
```

Zamiast przypisywania odwołania, operator ten nakazuje PHP utworzenie duplikatów wartości przechowywanych w zmiennej `$rasmus` i zapisanie ich w nowym obiekcie, który przypisywany jest do zmiennej `$zeev`. Oznacza to również, że `$rasmus` i `$zeev` są jednostkami niezależnymi, zatem wywołanie `$zeev->setName()` nie spowoduje zmian w `$rasmus`.

Jeżeli w programach PHP 4 powszechnie realizowane były operacje przekazywania i kopiowania przez wartość, można włączyć dyrektywę konfiguracyjną `zend.ze1_compatibility_mode`. Dzięki temu PHP 5 będzie klonował obiekty zamiast używać odwołań do nich.

Dyrektywa ta przywraca również niektóre mechanizmy PHP 4, które z PHP 5 zostały wyeliminowane. Na przykład, nie można już rzutować obiektów na liczby całkowite lub zmiennopozycyjne. W PHP 4 obiekty zawierające właściwości były rzutowane na wartość `1`, a obiekty bez właściwości — na wartość `0`.

Włączenie trybu zgodności może ułatwić przechodzenie na PHP 5, lecz nie powinno być rozwiązaniem długoterminowym. Zmniejsza ono bowiem przenośność aplikacji, a poza tym nie można współdzielić kodu między aplikacją, w której tryb zgodności został włączony, a taką, w której tryb ten jest wyłączony. Nowe witryny powinno się tworzyć pozostawiając domyślną wartość tej dyrektywy (`Off`).

Odśmiecanie

W niektórych językach, przede wszystkim w C, wymagane jest jawne każdorazowe pozyskiwanie od komputera pamięci, gdy trzeba utworzyć łańcuchy znaków czy struktury danych. Dopiero po zaalokowaniu pamięci można zapisać daną w zmiennej.

Na programiście spoczywa także obowiązek zwalniania pamięci, gdy używanie zmiennej zostanie zakończone. Dzięki temu komputer będzie mógł przydzielać tę pamięć innym zmiennym występującym w programie i zapobiegnie wyczerpaniu się pamięci operacyjnej. Rozwiązanie takie jest bardzo niewygodne.

PHP sam przeprowadza alokację pamięci. W momencie tworzenia egzemplarza obiektu pamięć jest przydzielana automatycznie, a w chwili usuwania obiektu pamięć zostaje zwolniona.

Proces czyszczenia obiektów nieużywanych to tak zwane **odśmiecanie**. Typ odśmiecania wykonywanego przez PHP to **zliczanie odwołań**.

Gdy tworzona jest nowa wartość — na przykład łańcuch znaków, liczba lub obiekt — PHP zapamięta jej istnienie i ustawi licznik odwołań na jeden. Będzie to oznaczać, że istnieje jedna kopia wartości. Od tej chwili PHP będzie śledził wartość, w odpowiednich momentach zwiększając lub zmniejszając wartość licznika.

Licznik zwiększy się o jeden, gdy utworzone zostanie odwołanie do wartości czy to przez przekazanie jej do funkcji przez odwołanie, czy przez przypisanie jej przez odwołanie do innej zmiennej. (Obiekty zawsze są przypisywane przez odwołanie, chyba że użyty zostanie operator clone; elementy nie będące obiektami są przypisywane przez odwołanie po użyciu operatora `=&`.) Wartość licznika zmniejszy się o jeden, gdy odwołanie do wartości zostanie usunięte. Ma to miejsce w momencie wyjścia z funkcji lub usunięcia zmiennej. Na przykład:

```
$rasmus1 = new Person;      // Nowy obiekt:                  Licznik odwołań = 1
$rasmus2 = $rasmus1;        // Skopiowanie przez odwołanie:  Licznik odwołań = 2
unset($rasmus1);            // Usunięcie odwołania:          Licznik odwołań = 1
sendEmailTo($rasmus2);      // Przekazanie przez odwołanie:
                            // W trakcie wykonywania funkcji:
                            //                               Licznik odwołań = 2
                            // Po zakończeniu wykonywania funkcji:
                            //                               Licznik odwołań = 1
unset($rasmus2);            // Usunięcie odwołania:          Licznik odwołań = 0
```

Gdy licznik osiągnie wartość zero, PHP będzie wiedział, że obiekt nie jest już nigdzie używany w programie, więc go usunie i zwolni pamięć. Zanim jednak to nastąpi, PHP wywoła destruktor obiektu, aby pozwolić programiście na wyczyszczenie innych zasobów otwartych w obiekcie.

Na końcu skryptu PHP wyczyści wszystkie pozostałe wartości, dla których wartość licznika odwołań będzie niezerowa.

Klasy

Aby zdefiniować klasę, należy użyć słowa kluczowego `class` i podać po nim nazwę klasy:

```
class Person {

}
```

Kod ten powoduje utworzenie klasy `Person`. Nie jest to klasa budząca szczególne uznanie, ponieważ brak w niej metod i właściwości. Wielkość liter w nazwach klas nie ma dla PHP żadnego znaczenia, dlatego nie można zadeklarować jednocześnie klas `Person` i `PERSON`.

Nazwy klasy używa się do tworzenia nowego egzemplarza obiektu:

```
$rasmus = new Person;
```

Aby ustalić klasę, z której wywodzi się obiekt, można użyć metody `get_class()`:

```
$person = new Person;
print get_class($person);
Person
```

Pomimo tego, że wielkość znaków w nazwach klas nie ma znaczenia, PHP 5 zapamiętuje ich wielkość. Różni się tym samym od PHP 4, który przekształcał wszystkie nazwy klas w małe litery. W PHP 4 funkcja `get_class()` wywołana na kopii obiektu `Person` zwróciłaby wartość `person`. PHP 5 natomiast zwróci prawidłową nazwę klasy.

Właściwości

Właściwości klasy wymienia się na jej początku:

```
class Person {
   public $name;
}
```

Kod ten spowoduje utworzenie właściwości publicznej o nazwie `name`. Właściwość publiczną można odczytywać oraz zapisywać do niej w dowolnym miejscu programu:

```
$rasmus = new Person;
$rasmus->name = 'Rasmus Lerdorf';
```

W PHP 4 właściwości są deklarowane inaczej — przy użyciu słowa kluczowego `var`. Składnia ta została zarzucona na korzyść `public`, zachowano jednak również wsteczną zgodność: `var` jest wciąż dozwolone. Zachowanie właściwości zadeklarowanej jako `public` oraz przy użyciu `var` jest identyczne.

Nigdy nie należy używać właściwości `public`. Odstępstwo od tej zasady sprawi, że bardzo łatwo będzie można naruszyć enkapsulację danych. Zamiast właściwości publicznych powinno się używać metod dostępu.

Aby móc używać właściwości od razu wewnątrz klasy, nie trzeba jej wcześniej oddzielnie deklarować. Na przykład:

```
$rasmus = new Person;
$rasmus->email = 'rasmus@php.net';
```

Spowoduje to przypisanie wartości `rasmus@php.net` do właściwości `email` zmiennej `$rasmus`. Jest to działanie prawidłowe pomimo tego, że `email` nie został wcześniej wymieniony w definicji klasy.

Mimo że nie ma takiej potrzeby, zawsze powinno się wcześniej deklarować właściwości. W przeciwnym razie właściwości te będą po pierwsze niejawnie potraktowane jako publiczne, co już nie jest pochwalane, a po drugie wcześniejsze zadeklarowanie właściwości zmusi do zastanowienia się nad najlepszym sposobem obsługi danych. Ponadto każdej osobie czytającej kod (włączając w to samego autora dwa miesiące później) łatwiej będzie przeczytać definicję klasy, ponieważ od razu widoczne będą wszystkie jej właściwości i nie trzeba będzie przedzierać się przez całą definicję klasy.

Metody

Metody definiuje się pod właściwościami. Są one deklarowane przy użyciu standardowej składni deklarowania funkcji:

```
class Person {
   public $name;

   public function setName($name) {
      $this->name = $name;
   }
}
```

Słowo kluczowe `public` oznacza, że metodę `setName()` może wywołać każdy. W PHP 4 nazw metod nie poprzedzało się identyfikatorem ich widzialności, jakim jest między innymi `public`. Dla zachowania wstecznej zgodności przyjmuje się, że tak zadeklarowane metody są publiczne.

W odróżnieniu od właściwości, metody o charakterze publicznym nie są niczym złym. Na przykład metody dostępu często deklaruje się jako publiczne.

Aby odwołać się do kopii obiektu wewnątrz klasy, należy użyć specjalnego słowa kluczowego `$this`. Na przykład:

```
public function setName($name) {
   $this->name = $name;
}
```

W kodzie tym metoda `setName()` przypisze właściwości `name` bieżącego obiektu wartość zmiennej `$name`, przekazanej do tej metody.

Należy zachować ostrożność i nie wstawiać przed nazwą właściwości znaku dolara, pisząc na przykład `$this->$name`. Taki zapis spowoduje, że PHP uzyska dostęp do właściwości wskazywanej przez wartość przechowywaną w zmiennej `$name`. Czasami jest to pożądany wynik, rzadko jednak dochodzi do takich sytuacji.

PHP 4 nie zapobiega przypisywaniu obiektowi `$this` nowego obiektu:

```
public function load($object) {
   $this = $object;
}
```

W PHP 5 operacja taka jest już niedozwolona — można zmieniać jedynie właściwości obiektu. Próba przypisania `$this` nowej wartości spowoduje wygenerowanie błędu:

```
PHP Fatal error: Cannot re-assign $this
```

Ograniczenia dostępu

Aby zapobiec uzyskiwaniu dostępu do właściwości lub metody z zewnątrz klasy, należy użyć słowa kluczowego `private`:

```
class Person {
   private $name;

   public function setName($name) {
      $this->name = $name;
   }
}

$rasmus = new Person;
$rasmus->setName('Rasmus Lerdorf');
print $rasmus->name;
Fatal error: Cannot access private property Person::$name... on line 13
```

Gdy właściwość `name` jest zadeklarowana jako `private`, nie można uzyskać do niej dostępu spoza klasy. Cały czas jednak można operować na niej wewnątrz metody `setName()`, ponieważ jest to metoda wewnętrzna. Nie można na przykład wykonać następującej instrukcji:

```
print $rasmus->name;
```

Spowoduje ona powstanie błędu krytycznego, dlatego konieczne jest zaimplementowanie metody `getName()`:

```
class Person {
   private $name;

   public function setName($name) {
      $this->name = $name;
   }

   public function getName() {
      return $this->name;
   }
}

$rasmus = new Person;
$rasmus->setName('Rasmus Lerdorf');
print $rasmus->getName();
Rasmus Lerdorf
```

Ten kod zadziała już zgodnie z oczekiwaniami i zwróci wartość **`Rasmus Lerdorf`**.

Metody prywatne deklaruje się, umieszczając przed słowem `function` słowo `private`:

```
class Person {
   private $email;

   public function setEmail($email) {
      if ($this->validateEmail($email)) {
          $this->email = $email;
      }
   }

   private function validateEmail($email) {
       // weryfikacja poprawności adresu pocztowego
       // wyrażenie regularne
       // pominięto dla uproszczenia przykładu
   }
}

$rasmus = new Person;
$rasmus->setEmail('rasmus@ph.net');
```

W kodzie tym zadeklarowano dwie metody: publiczną i prywatną. Metoda publiczna `setEmail()` służy do ustawiania osobistego adresu poczty elektronicznej, natomiast metoda prywatna `validateEmail()` jest używana przez klasę wewnętrznie do sprawdzania, czy podany adres jest prawidłowy. Metoda ta nie ma znaczenia dla użytkownika końcowego, dlatego została zadeklarowana jako `private`.

W powyższym przykładzie widać także, jak wewnątrz klasy uzyskuje się dostęp do metody. Składnia przypomina składnię stosowaną w celu uzyskania dostępu do właściwości klasy. W celu reprezentacji obiektu należy użyć `$this`, jak uczyniono to wewnątrz `setEmail()`:

```
public function setEmail($email) {
   if ($this->validateEmail($email)) {
      $this->email = $email;
   }
}
```

Kod ten wywołuje metodę `validateEmail()` klasy `Person` i przekazuje do niej zmienną `$email`. Wywołanie to pojawia się w metodzie zdefiniowanej w tej samej klasie, dlatego zadziała nawet pomimo tego, że `validateEmail()` zadeklarowano jako `private`.

Konstruktory i destruktory

Konstruktory obiektów działają w PHP 5 tak samo jak w PHP 4, lecz w PHP 5 wprowadzono nową konwencję nazewniczą. W PHP 4 konstruktor obiektu ma nazwę taką samą jak jego klasa:

```
class Database {
   function Database($host, $user, $password) {
      $this->handle = db_connect($host, $user, $password);
   }
}

$db = new Database('db.przyklad.com', 'web', 'jsd6w@2d');
```

Utworzenie nowego egzemplarza klasy `Database` spowoduje, że PHP wywoła metodę `Database()`.

Aby wyznaczyć konstruktor obiektu w PHP 5, należy nadać metodzie nazwę `__construct()`:

```
class Database {
   function __construct($host, $user, $password) {
      $this->handle = db_connect($host, $user, $password);
   }
}

$db = new Database('db.przyklad.com', 'web', 'jsd6w@2d');
```

Aby ułatwić przejście z PHP 4, założono, że jeżeli PHP 5 nie będzie mógł znaleźć wewnątrz hierarchii obiektów metody o nazwie `__construct()`, powróci on do konwencji nazewniczej konstruktorów używanej w PHP 4 i ponowi poszukiwania. W PHP 4 konstruktor ma taką samą nazwę jak klasa, a zatem o ile w kodzie nie występuje metoda o nazwie `__constructor()` wykonująca inne zadania, istniejący kod nie powinien generować błędów w PHP 5 (przyczyny dokonania takiej zmiany zostaną wyjaśnione w punkcie „Konstruktory" w dalszej części rozdziału).

W przypadku destruktorów trudno mówić o jakiejkolwiek zgodności wstecznej, ponieważ w PHP 4 w ogóle one nie występują. Nie oznacza to jednak, że programiści nie podejmowali prób ich stworzenia przy użyciu innych mechanizmów języka. Jeśli w kodzie napisanym wcześniej emulowano destruktory, najlepiej będzie przenieść ten kod do PHP 5, ponieważ udostępniane w nim destruktory są bardziej wydajne i łatwiejsze w użyciu.

W PHP 4 można naśladować destruktory definiując metodę, która będzie miała odgrywać ich rolę, a następnie rejestrując ją w funkcji `register_shutdown_function()` jako tę, którą PHP powinien wywołać na końcu skryptu. Listing 2.2 zawiera odpowiedni przykład:

Listing 2.2. Naśladowanie destruktorów w PHP 4

```
register_shutdown_function('destruct');
$GLOBALS['objects_to_destroy'] = array();

function destruct() {
   foreach($GLOBALS['objects_to_destroy'] as $obj) {
      $obj->destruct();
   }
}

class Database {
   function Database($host, $user, $password) {
      $this->handle = db_connect($host, $user, $password);
      $GLOBALS['objects_to_destroy'][] = &$this;
   }

   function destruct() {
      db_close($this->handle); // zamknięcie połączenia z bazą danych
   }
}
```

PHP udostępnia specjalną funkcję o nazwie `register_shutdown_function()`, która jest wywoływana bezpośrednio przed zakończeniem skryptu. Funkcji tej można użyć w celu zagwarantowania, że przed wykonaniem własnych operacji kończących PHP uruchomi kod wskazany przez programistę.

Kod z listingu 2.2 tak ustawia cały mechanizm, by PHP wywoływał funkcję `destruct()`. Przechodzi ona kolejno przez listę obiektów do zniszczenia przechowywaną w zmiennej globalnej `$objects_to_destroy` i na każdym z nich wywołuje metodę `destruct()`.

Jeżeli dany obiekt wymaga istnienia destruktora, musi zostać dołączony do tablicy `$objects_to_destroy` oraz posiadać zaimplementowaną metodę `destruct()`. Metoda ta powinna zawierać dowolny kod konieczny do wyczyszczenia zasobów, które były wykorzystywane w czasie tworzenia i używania obiektu.

W powyższym przykładzie klasa `Database` dołącza się w konstruktorze, wykonując instrukcję `$GLOBALS['objects_to_destroy'][] = &$this;`. Gwarantuje to, że wszystkie obiekty zostaną odpowiednio obsłużone. Metoda `destruct()` tej klasy wywołuje metodę `db_close()` zamykającą połączenie z bazą danych.

W wielu przypadkach, takich jak zamykanie połączenia z bazą danych i odblokowanie plików, PHP wykona odpowiednie czynności automatycznie. Jednak zostaną one zrealizowane dopiero w momencie zakończenia wykonywania skryptu. Dobrym pomysłem będzie zatem zwolnienie tych zasobów samodzielnie przy użyciu destruktora. Najlepiej jest zwalniać je tak szybko jak to możliwe, ponieważ inne programy mogą wymagać dostępu do bazy danych lub do zablokowanego pliku.

W czasach, gdy większość skryptów PHP była krótka i działała szybko, zostawienie czyszczenia zasobów na barkach PHP nie było wielkim problemem, ponieważ czas między zakończeniem używania zasobu oraz zakończeniem wykonywania skryptu był bardzo krótki. Teraz jednak, gdy PHP jest używany w wierszu poleceń i wykonuje zadania o wiele bardziej złożone, skrypty działające przez dłuższy czas stały się normą, a więc i waga tego problemu wzrosła.

Nietrudno zauważyć, że zaimplementowanie destruktora przy użyciu funkcji `register_shutdown_function()` w żaden sposób nie pozwala zaoszczędzić czasu, ponieważ destruktor ten zostanie wywołany dopiero w chwili zakończenia skryptu. Jest to jeden z najważniejszych elementów odróżniających emulację destruktorów w PHP 4 od destruktorów w PHP 5.

W PHP 5 obiekty są niszczone wówczas, gdy nie będą już więcej używane, zatem połączenia są zwalniane znacznie wcześniej. Ponadto sposób implementacji stosowany w PHP 4 daleki jest od czystości i obiektowości. Do śledzenia obiektów używa się w nim zmiennych globalnych oraz funkcji globalnych, przez co łatwo jest naruszyć cały mechanizm nadpisując tablicę.

Na szczęście w PHP 5 destruktory zostały zaimplementowane w samym języku, dzięki czemu PHP automatycznie sprawdza, które obiekty posiadają destruktory, i wywołuje je od razu w momencie, w którym ich używanie dobiegnie końca. Może to następować już na długo przed zakończeniem samego programu, a więc zasoby takie jak połączenia z bazą danych czy blokady na plikach nie będą utrzymywane przez cały czas wykonywania skryptu, lecz zostaną zwolnione przy pierwszej sposobności.

Podobnie jak konstruktory, destruktory również mają w PHP 5 stałą nazwę `__destruct()`. Jako że nie są one wywoływane ręcznie, nie można przekazywać do nich żadnych parametrów. Jeżeli w destruktorze wymagana jest jakakolwiek informacja o obiekcie, trzeba ją przechowywać we właściwości:

```
// Destruktor w PHP 5
class Database {
   function __destruct() {
      db_close($this->handle); // zamknięcie połączenia z bazą danych
   }
}
```

Destruktory są już mechanizmem funkcjonującym na poziomie języka, a więc nie ma potrzeby używania funkcji `register_shutdown_global()`. Wszystkie konieczne operacje są wykonywane automatycznie.

Nie można zakładać, że PHP zniszczy obiekty w z góry ustalonej kolejności. W destruktorze nie powinno się zatem odwoływać do innych obiektów, ponieważ mogły one już ulec zniszczeniu. Czynność taka nie spowoduje załamania aplikacji, lecz sam kod będzie się wówczas zachowywał w sposób nieprzewidywalny i generował błędy.

Mechanizmy pośrednie wobec klas

W poprzednim punkcie opisano ograniczenia mechanizmów obiektowych dostępnych w PHP 4. W tym punkcie natomiast przedstawionych zostanie kilka mechanizmów stanowiących nowość w PHP 5: interfejsy, wskazywanie typów oraz metody i właściwości statyczne.

Interfejsy

W programowaniu zorientowanym obiektowo obiekty muszą ze sobą współpracować. Powinna zatem istnieć możliwość wymuszania na klasie (lub klasach), by implementowała ona metody niezbędne do prawidłowej interakcji z innymi elementami w systemie.

Na przykład aplikacja typu e-commerce powinna posiadać odpowiedni zestaw informacji o każdym towarze wystawianym na sprzedaż. Towary te mogą być reprezentowane przez odpowiednie klasy: `Book`, `CD`, `DVD` i tak dalej. Musimy jednak mieć także pewność, że aplikacja będzie potrafiła znaleźć nazwę, cenę i numer identyfikacyjny każdego towaru bez względu na jego rodzaj.

Mechanizmem, który wymusza na klasie obsługę tego samego zestawu metod, jest **interfejs**. Definiuje się go podobnie jak definiuje się klasę:

```
interface Sellable {
   public function getName();
   public function getPrice();
   public function getID();
}
```

Zamiast słowa kluczowego `class` w definicji interfejsu używa się słowa kluczowego `interface`. Wewnątrz interfejsu definiuje się natomiast prototypy metod, lecz nie podaje się ich implementacji.

Powyższy kod utworzy interfejs o nazwie `Sellable`. Każda klasa implementująca interfejs `Sellable` musi implementować wskazane w nim trzy metody: `getName()`, `getPrice()` i `getID()`.

O klasie obsługującej wszystkie metody interfejsu mówi się, że **implementuje interfejs**. W definicji tej klasy należy wskazać, że implementuje ona odpowiedni interfejs:

```
class Book implements Sellable {

   public function getName() { ... }
   public function getPrice() { ... }
   public function getID() { ... }
}
```

Jeżeli klasa nie będzie zawierać implementacji wszystkich metod wymienionych w interfejsie lub będzie implementować je z innym prototypem, PHP wygeneruje błąd krytyczny.

Klasa może implementować dowolną liczbę interfejsów. Można na przykład utworzyć interfejs `Listenable` opisujący sposoby zakupu towaru z zawartością dźwiękową. W takim przypadku klasy `CD` i `DVD` implementowałyby także interfejs `Listenable`, lecz klasa `Book` już nie.

Jeżeli używa się interfejsów, należy pamiętać, by deklarować klasy jeszcze przed tworzeniem egzemplarzy obiektów. W PHP 4 kod można układać w sposób dowolny, bo i tak PHP sam znajdzie definicję klasy.

W PHP 5 również tak się stanie w większości przypadków, jednak gdy klasa implementuje interfejs, PHP 5 może czasami zachowywać się niewłaściwie. Deklarowanie takiej klasy przed tworzeniem egzemplarzy obiektów nie jest wymagane by nie unieruchamiać już istniejących aplikacji, lecz najlepiej jest nie polegać na PHP w tym względzie.

Wskazywanie typów

Kolejnym sposobem wymuszania kontroli obiektów jest używanie **wskazań typów**. Wskazanie typu jest mechanizmem informującym PHP, że obiekt przekazywany do metody powinien być obiektem konkretnej klasy.

W PHP 4 trzeba samodzielnie sprawdzać, czy argument ma właściwy typ. W efekcie wewnątrz kodu często trzeba wywoływać funkcje `get_class()` i `is_array()`.

Aby zdjąć ten ciężar z programistów, PHP 5 sam bierze na siebie zadanie polegające na sprawdzaniu typu. Opcjonalnie w prototypie funkcji i metody można wskazać nazwę klasy. Rozwiązanie takie działa jednak tylko w przypadku klas, a dla zmiennych innych typów już nie. Nie można na przykład wymusić, by argument był tablicą.

Aby wymusić, by pierwszy argument metody `add()` klasy `AddressBook` był typu `Person`, należy napisać:

```
class AddressBook {

   public function add(Person $person) {
      // dodaje $person do książki adresowej
   }
}
```

Wówczas jeśli `add()` zostanie wywołana z łańcuchem znaków, zwrócony zostanie błąd krytyczny:

```
$book = new AddressBook;

$person = 'Rasmus Lerdorf';
$book->add($person);
PHP Fatal error: Argument 1 must be an object of class Person in...
```

Umieszczenie wskazania typu `Person` w pierwszym argumencie deklaracji funkcji da taki sam efekt, jak dodanie do funkcji następującego kodu PHP:

```
public function add($person) {
   if (!($person instanceof Person)) {
      die("Argument 1 must be an instance of Person");
   }
}
```

Operator `instanceof` sprawdza, czy obiekt jest egzemplarzem konkretnej klasy. Kod ten zapewnia, że zmienna `$person` jest typu `Person`.

W PHP 4 nie ma operatora `instanceof`. Zamiast niego trzeba używać funkcji `is_a()`, z której w PHP 5 zrezygnowano.

Wskazywanie typów ma również tę dodatkową zaletę, że prowadzi do zintegrowania dokumentacji API bezpośrednio z klasą. Jeżeli zobaczymy, że konstruktor klasy przyjmuje argument typu `Event`, od razu wiadomo, jaką metodę należy wywołać. Wiadomo ponadto, że kod i „dokumentacja" muszą być ze sobą zawsze zsynchronizowane, ponieważ wymóg ten zawiera się bezpośrednio w definicji klasy.

Możliwość wskazywania typów wiąże się jednak z obniżeniem elastyczności. Nie istnieje sposób, by umożliwić przyjmowanie parametru o więcej niż jednym typie, a to ogranicza z kolei możliwości projektowania hierarchii obiektów.

Ponadto konsekwencje nieprzestrzegania wskazań typów są dość drastyczne: wykonywanie skryptu zostaje przerwane i zwracany jest błąd krytyczny. W aplikacjach dla sieci WWW programiści zwykle dążą do objęcia większej kontroli nad sposobem obsługi błędów, aby obsługiwać je w sposób bardziej elegancki. Dzięki zaimplementowaniu w metodach własnego mechanizmu sprawdzania typów, można wyświetlać stronę zawierającą komunikat o błędzie.

Warto na koniec wspomnieć, że w odróżnieniu od niektórych języków, w PHP nie można wskazywać typu zwracanych wartości, a zatem nie można wymusić, by konkretna funkcja zawsze zwracała obiekt konkretnego typu.

Metody i właściwości statyczne

Czasami może zaistnieć potrzeba zdefiniowania w obiekcie szeregu metod, a jednocześnie zapewnienia sobie możliwości ich uruchamiania bez zawracania sobie głowy tworzeniem egzemplarza obiektu. W PHP 5 można zadeklarować metodę `static`, dzięki czemu można ją wywoływać bezpośrednio:

```
class Format {
   public static function number($number, $decimals = 2,
                                 $decimal = ',', $thousands = '.') {
      return number_format($number, $decimals, $decimal, $thousands);
   }
}

print Format::number(1234.567);
1,234.57
```

Metody statyczne nie wymagają istnienia kopii obiektu, dlatego zamiast obiektu można używać w ich przypadku nazwy klasy. Przed nazwą klasy nie należy umieszczać znaku dolara (`$`).

Do metod statycznych nie odwołuje się poprzez symbol strzałki (`->`), lecz za pośrednictwem dwóch dwukropków (`::`) — dzięki temu PHP wie, że ma do czynienia z metodą statyczną. Dlatego w powyższym przykładzie dostęp do metody `number()` klasy `Format` jest uzyskiwany poprzez `Format::number()`.

Sposób formatowania liczby nie zależy od jakichkolwiek właściwości ani metod obiektu, zatem sensownie będzie zadeklarować tę metodę jako `static`. Dzięki temu, na przykład w aplikacji koszyka na zakupy w prosty sposób można formatować cenę towarów, pisząc tylko jeden wiersz kodu i cały czas używając przy tym obiektu zamiast funkcji globalnej.

Metody statyczne nie operują na konkretnej kopii klasy, w której je zdefiniowano. PHP nie „tworzy" obiektu tymczasowego, który byłby używany w trakcie wykonywania kodu metody. Wewnątrz metody statycznej nie można zatem odwoływać się do `$this`, ponieważ nie ma żadnej zmiennej `$this`, do której można by się odwołać. Wywołanie metody statycznej nie różni się od wywołania zwykłej funkcji.

W PHP 5 dostępne są również tak zwane **właściwości statyczne**. Każda kopia klasy współdzieli tego typu właściwości. Właściwości statyczne odgrywają zatem rolę zmiennych globalnych w przestrzeni nazw klasy.

Właściwości statycznej można na przykład użyć do współdzielenia połączenia z bazą danych przez wiele różnych obiektów `Database`. Aby nie obniżać wydajności, powinno się unikać tworzenia nowego połączenia za każdym razem, gdy tworzony jest egzemplarz `Database`, lecz za pierwszym razem negocjować połączenie, a w każdej dodatkowej kopii klasy ponownie je wykorzystywać:

```
class Database {
   private static $dbh = NULL;

   public function __construct($server, $username, $password) {
      if (self::$dbh == NULL) {
          self::$dbh = db_connect($server, $username, $password);
      } else {
          // ponowne wykorzystanie istniejącego połączenia
      }
   }
}
```

```
$db = new Database('db.przyklad.com', 'web', 'jsd6w@2d');
// wykonanie zapytań

$db2 = new Database('db.przyklad.com', 'web', 'jsd6w@2d');
// wykonanie zapytań dodatkowych
```

Podobnie jak w przypadku metod statycznych, również w odniesieniu do właściwości statycznych używa się zapisu z dwoma średnikami. Aby odwołać się do właściwości statycznej wewnątrz klasy, należy użyć specjalnego prefiksu `self`. Prefisk `self` pełni taką samą rolę dla statycznych metod i właściwości jak `$this` dla właściwości i metod utworzonej kopii klasy.

W konstruktorze użyto zapisu `self::$dbh`, aby uzyskać dostęp do statycznej właściwości `connection`. Po utworzeniu kopii `$db` `dbh` ma cały czas wartość `NULL`, zatem konstruktor wywoła metodę `db_connect()`, aby wynegocjować nowe połączenie z bazą danych.

Gdy tworzone będzie `$db2`, czynności te nie zostaną już wykonane, ponieważ wcześniej do `dbh` przypisano już uchwyt do bazy danych.