helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

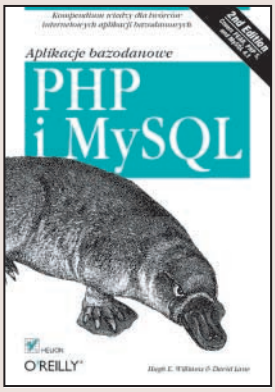
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

PHP i MySQL. Aplikacje bazodanowe



Autorzy: Hugh E. Williams, David Lane
Tłumaczenie: Michał Dadan (rozdz. 1 – 8, 10), Paweł Gonera (rozdz. 9, 16 – 20, dod. A – H), Daniel Kaczmarek (rozdz. 11 – 15)
ISBN: 83-7361-671-3
Tytuł oryginału: Web Database Applications with PHP and MySQL
Format: B5, stron: 792

Książka „PHP i MySQL. Aplikacje bazodanowe” jest przeznaczona dla tych, którzy tworzą lub zamierzają tworzyć witryny WWW oparte na technologii PHP i MySQL. Opisano w niej reguły i techniki wykorzystywane przy tworzeniu małych i średnich aplikacji bazodanowych wykorzystywanych do przechowywania danych, odczytywania ich i zarządzania nimi. Przedstawia zasady pracy z bazami danych. Pokazuje, jak śledzić poczynania użytkowników za pomocą sesji, pisać bezpieczny kod, oddzielać go od warstwy prezentacyjnej i uniezależniać go od wyboru bazy danych. Opisuje również techniki generowania raportów i obsługi błędów oraz zaawansowane zagadnienia związane z bazami danych i programowaniem zorientowanym obiektowo.

- Typowe modele architektury aplikacji bazodanowych
- Język PHP — podstawowe wiadomości
- Programowanie zorientowane obiektowo w PHP5
- Język SQL i baza danych MySQL
- Biblioteka PEAR
- Kontrola poprawności wprowadzanych danych z wykorzystaniem PHP i JavaScript
- Mechanizmy bezpieczeństwa w aplikacjach bazodanowych
- Wdrażanie aplikacji
- Generowanie raportów
- Przykład praktyczny — internetowy sklep z winami

Wiadomości zawarte w tej książce pomogą każdemu programiście stworzyć sklep internetowy, portal lub system zarządzania treścią.

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

Spis treści

Wstęp **7**

1. Aplikacje bazodanowe a Internet **17**
Sieć WWW 18
Architektury trójwarstwowe 19

2. Język skryptowy PHP **33**
Wprowadzenie do PHP 33
Instrukcje rozgałęziające i wyrażenia warunkowe 45
Pętle 49
Funkcje 52
Praca z typami 53
Funkcje definiowane przez użytkownika 58
Praktyczny przykład 68

3. Tablice, łańcuchy i zaawansowane operacje na danych **71**
Tablice 71
Łańcuchy 89
Wyrażenia regularne 99
Daty i godziny 108
Liczby całkowite i zmiennopozycyjne 114

4. Wprowadzenie do programowania zorientowanego obiektowo w PHP 5 ..... **119**
Klasy i obiekty 119
Dziedziczenie 133
Zgłaszanie i obsługiwanie wyjątków 140

5. SQL i MySQL **143**
Podstawy baz danych 143
Interpreter poleceń MySQL 149
Zarządzanie bazami danych i tabelami 151
Wstawianie, uaktualnianie i usuwanie danych 157
Zapytania z wyrażeniem SELECT 161
Złączenia 169
Praktyczny przykład: dodawanie nowego wina 176

6. Kierowanie zapytań do baz danych 179
Przesyłanie zapytań do baz MySQL z poziomu PHP 180
Przetwarzanie informacji wprowadzanych przez użytkowników 195
Opis funkcji biblioteki MySQL 214

7. PEAR 225
Pierwsze spojrzenie 225
Podstawowe składniki 226
Pakiety 236

8. Umieszczanie danych w internetowych bazach danych 257
Wstawianie, uaktualnianie i usuwanie informacji z baz danych 257
Problemy z zapisywaniem informacji w bazach danych 275

9. Weryfikacja danych za pomocą PHP i języka JavaScript 291
Zasady kontroli poprawności i raportowania błędów 291
Weryfikacja po stronie serwera za pomocą PHP 294
JavaScript i kontrola poprawności po stronie klienta 311

10. Sesje 339
Wprowadzenie do zarządzania sesjami 340
Zarządzanie sesjami w PHP 341
Przykład praktyczny: stosowanie sesji przy weryfikacji danych 348
Kiedy należy stosować sesje? 357
API zarządzania sesjami i konfiguracja sesji 360

11. Uwierzytelnianie i bezpieczeństwo 371
Uwierzytelnianie HTTP 371
Uwierzytelnianie HTTP w PHP 375
Uwierzytelnianie na podstawie formularza 386
Ochrona danych w sieci WWW 398

12. Błędy, debugowanie i wdrażanie 403
Błędy 403
Najczęstsze błędy programistyczne 408
Własne mechanizmy obsługi błędów 413

13. Raporty 423
Tworzenie raportu 423
Tworzenie dokumentu PDF 428
Instrukcja PDF-PHP 440

14. Zaawansowane programowanie obiektowe w PHP 5 457
Korzystanie z hierarchii klas 457
Wskazanie typu klasy 461
Klasy abstrakcyjne i interfejsy 462
Przykład: kalkulator kosztów transportu 467

15. Zaawansowany SQL 477
Analiza przy użyciu polecenia SHOW 478
Zapytania zaawansowane 479
Operacje na danych i bazach danych 494
Funkcje 502
Automatyzacja wykonywania zapytań 510
Typy tabel 513
Kopie zapasowe i ich odtwarzanie 519
Zarządzanie użytkownikami i uprawnieniami 524
Dostrajanie serwera MySQL 528

16. Sieciowa winiarnia „Hugh i Dave". Analiza przypadku 539
Wymagania systemowe i funkcjonalne 540
Omówienie aplikacji 542
Komponenty współdzielone 547

17. Zarządzanie kontami klientów 575
Przegląd kodu 576
Kontrola poprawności danych klienta 579
Formularz klienta 582

18. Koszyk na zakupy 587
Przegląd kodu 588
Strona domowa sieciowej winiarni 589
Implementacja koszyka 594

19. Zamawianie i wysyłka w sieciowej winiarni 607
Przegląd kodu 607
Dane karty kredytowej i instrukcje wysyłki 609
Realizacja zamówienia 612
Potwierdzenia z poziomu strony HTML oraz przez e-mail 618

20. Wyszukiwanie i autoryzacja w sieciowej winiarni 629
Przegląd kodu 630
Przeglądanie i wyszukiwanie 634
Autoryzacja 643

A Przewodnik instalacji w systemie Linux 651

B Przewodnik instalacji w systemie Microsoft Windows 671

C Przewodnik instalacji w systemie Mac OS X 681

D Protokoły sieciowe 697

E Modelowanie i projektowanie relacyjnych baz danych 709

F Zarządzanie sesjami w warstwie bazy danych 727

G Zasoby 741

H Ulepszona biblioteka MySQL 745

Skorowidz 757

ROZDZIAŁ 4.

Wprowadzenie do programowania zorientowanego obiektowo w PHP 5

Idea programowania zorientowanego obiektowo ma już parę dziesięcioleci. Zyskała ona taką popularność, że obecnie stosuje się ją niemal we wszystkich językach programowania. Powody takiego stanu rzeczy łatwo zrozumieć gdy zacznie się wykorzystywać tak wygodne rozwiązania, jak pakiety dodatkowych funkcji dla PHP. Jeden z nich, o nazwie PEAR, opisujemy w rozdziale 7. Wiele z tych funkcji definiuje własne obiekty, dzięki którym cała ich funkcjonalność jest udostępniana w bardzo przystępnej formie. Aby móc korzystać z pakietów i z tak zwanych wyjątków, które ułatwiają obsługę błędów, trzeba poznać podstawy programowania zorientowanego obiektowo. Możliwe, że czytelnik dojdzie nawet do wniosku, że będzie chciał wykorzystywać obiekty w swoim własnym kodzie. Ten rozdział stanowi wprowadzenie w świat obiektów, a ich zaawansowane możliwości opisujemy w rozdziale 14.

Założenia i techniki, o których tu piszemy, odnoszą się przede wszystkim do nowej, znacznie rozbudowanej wersji PHP 5. Wiele z nich w PHP 4 wygląda tak samo, ale nie wszystkie. Dlatego w dalszej części rozdziału piszemy wyraźnie, co można zrobić w konkretnej wersji języka.

Klasy i obiekty

Główną ideą stojącą za programowaniem zorientowanym obiektowo jest to, że dane można umieścić razem z funkcjami w wygodnych pojemnikach zwanych *obiektami*. Na przykład w rozdziale 7. powiemy jak można nadać kilku witrynom taki sam wygląd, korzystając z obiektu nazywanego szablonem (ang. *template*). W naszym kodzie PHP będziemy mogli odwoływać się do niego poprzez dowolną zmienną. Na potrzeby tego przykładu przyjmijmy, że nazwiemy ją `$template`. Wszystkie szczegóły implementacyjne poszczególnych szablonów są przed nami ukryte. Wystarczy jedynie, że wczytamy odpowiedni pakiet i umieścimy w kodzie PHP wyrażenie tego typu:

```
$template = new HTML_Template_IT("./templates");
```

Jak sugeruje użycie operatora `new` (nowy), właśnie utworzyliśmy nowy obiekt. Będzie się on nazywał `$template` i został utworzony przez pakiet `HTML_Template_IT`, którego kodu wcale nie musimy znać! Mając obiekt `$template` możemy już korzystać z całej funkcjonalności oferowanej przez pakiet `HTML_Template_IT`.

Po przeprowadzeniu wielu różnych operacji na tym obiekcie możemy wyświetlić go na stronie WWW za pomocą polecenia:

```
$template->show();
```

Warto przyjrzeć się składni tego polecenia. Jak sugeruje użycie nawiasów, `show()` jest funkcją. Jest ona jednak powiązana za pomocą operatora `->` ze zmienną `$template`. Gdy funkcja `show()` zostanie wywołana, pobierze ona dane przechowywane w obiekcie `$template` i to na ich podstawie wyliczy wyniki. Innymi słowy, funkcja `show()` jest wywoływana na obiekcie `$template`.

To, jakie funkcje wywołujemy, zależy od tego, jakie są dostępne w danym pakiecie. Na przykład pakiet `HTML_Template_IT` udostępnia funkcję `show()`, co oznacza, że można ją wywoływać na obiektach `HTML_Template_IT` takich jak `$template`. W tradycyjnym języku programowania zorientowanego obiektowo funkcję `show()` nazywamy *metodą* lub *funkcją składową* obiektu `HTML_Template_IT`.

Mówimy, że `HTML_Template_IT` jest *klasą*, ponieważ możemy wykorzystać ją do utworzenia dowolnej liczby takich samych obiektów-szablonów. Tak więc obiekt `$template` jest obiektem klasy `HTML_Template_IT`[1].

Wiadomo już więc, jak tworzyć obiekty klas zdefiniowanych w pakietach. Jednak, aby czytelnik lepiej zrozumiał naturę obiektów, pokażemy teraz, jak można zdefiniować własną klasę. Na listingu 4.1 przedstawiamy prostą klasę wymyśloną na potrzeby tego rozdziału, o nazwie `UnitCounter`. Oferuje ona dwie banalne funkcje — można wykorzystywać ją do zliczania określonych przedmiotów oraz do wyliczenia ich sumarycznej masy. W dalszej części tego rozdziału, a także w rozdziale 14., będziemy wykorzystywali tę klasę w połączeniu z innymi klasami i utworzymy prosty kalkulator obliczający należność za przewóz towarów.

Listing 4.1. Definicja nowej klasy UnitCounter

```
<?php

// Definicja klasy UnitCounter
class UnitCounter
{
   // Zmienne składowe
   var $units = 0;
   var $weightPerUnit = 1.0;

   // Dodaj $n do całkowitej liczby sztuk. Niech domyślną wartością $n będzie 1.
   function add($n = 1)
   {
      $this->units = $this->units + $n;
   }

   // Funkcja składowa obliczająca całkowitą masę

   function totalWeight()
   {
      return $this->units * $this->weightPerUnit;
   }
}

?>
```

[1] Wyobraź sobie, że klasa to foremka do piasku, a obiekty to babki, które nią wyciskasz — *przyp. tłum.*

Jak widać na listingu 4.1, klasę `UnitCounter` definiujemy posługując się słowem kluczowym `class`. Klasa ta ma dwie zmienne składowe — `$units` i `$weightPerUnit` i dwie funkcje składowe — `add()` i `totalWeight()`. Łącznie wspomniane funkcje i zmienne nazywamy *składowymi* klasy `UnitCounter`.

Definicja klasy określa, w jaki sposób określona funkcjonalność ma zostać powiązana z danymi. Funkcje i zmienne składowe nabierają znaczenia dopiero w kontekście klasy, której są częścią. Definicja klasy pokazana na listingu 4.1 nie powoduje wykonania żadnych działań ani zwrócenia żadnych wyników. Zamiast tego tworzy ona nowy typ danych, z którego będzie można korzystać w skrypcie PHP. W praktyce możemy umieścić tę definicję klasy w zewnętrznym pliku, który będziemy dołączać do każdego skryptu, w którym będzie ona potrzebna.

Aby móc odwoływać się do zmiennych i funkcji składowych zdefiniowanych w klasie, musimy utworzyć *obiekt* tej klasy. Podobnie jak dane innych typów, takich jak liczby całkowite, łańcuchy czy tablice, obiekty można przypisywać do zmiennych. Jednak w przeciwieństwie do zmiennych innych typów, obiekty tworzy się za pomocą operatora `new`. Oto jak można utworzyć obiekt klasy `UnitCounter` i przypisać go do zmiennej:

```
// Utwórz nowy obiekt UnitCounter
$bottles = new UnitCounter;
```

Inaczej niż w przypadku nazw zmiennych, PHP nie rozróżnia dużych i małych liter występujących w nazwach klas. Mimo iż my przyjęliśmy zasadę, aby zaczynać je wielką literą, tak naprawdę `UnitCounter`, `unitcounter` i `UNITCOUNTER` to jedna i ta sama klasa.

Po utworzeniu nowego obiektu `UnitCounter` i przypisaniu go do zmiennej `$bottles` możemy korzystać z jego funkcji i zmiennych składowych. Dostęp do składowych obiektu, zarówno funkcji jak i zmiennych, odbywa się poprzez operator `->`. Do zmiennej składowej `$units` możemy odwołać się poprzez wyrażenie `$bottles->units` i traktować ją dokładnie tak samo jak każdą inną zmienną:

```
// Ustaw licznik na dwa tuziny butelek
$bottles->units = 24;

// Wyświetl "Są 24 sztuki"
print "Są {$bottles->units} sztuki";
```

Chcąc umieścić wartość przechowywaną w zmiennej składowej obiektu wewnątrz łańcucha ujętego w cudzysłów, musimy użyć nawiasów klamrowych. Łańcuchy i nawiasy klamrowe omówiliśmy w rozdziale 2.

Do operowania wartością zmiennej `$bottles` możemy użyć funkcji składowej `add()`, wywołując ją poprzez wyrażenie `$bottles->add()`. Poniższy fragment kodu zwiększa wartość przechowywaną w zmiennej `$bottles->units` o `3`:

```
// Dodaj trzy butelki
$bottles->add(3);
// Wyświetl "Jest 27 sztuk"
print "Jest {$bottles->units} sztuk";
```

Możemy utworzyć kilka obiektów tej samej klasy. Na przykład poniższy fragment kodu tworzy dwa obiekty `UnitCounter` i przypisuje je do dwóch różnych zmiennych:

```
// Utwórz dwa obiekty UnitCounter
$books = new UnitCounter;
$cds = new UnitCounter;
```

```
// Dodaj po parę sztuk
$books->add(7);
$cds->add(10);

// Wyświetl "7 książek i 10 płyt"
print "{$books->units} książek i {$cds->units} płyt";
```

Obie zmienne, `$books` i `$cd`, odnoszą się do obiektów `UnitCounter`, ale każdy z tych obiektów jest niezależny od pozostałych.

Zmienne składowe

Zmienne składowe klas można deklarować zarówno w PHP 4, jak i w PHP 5.

Ich definicje umieszcza się w definicjach klas i poprzedza się je słowem kluczowym `var`. W tym miejscu można też stosować słowa kluczowe `private` i `protected`, o których piszemy w dalszej części rozdziału. Zadaniem zmiennych składowych jest trzymanie danych przechowywanych w obiekcie.

W definicji klasy można określić początkową wartość, jaką ma przyjąć dana zmienna składowa. Na przykład w klasie `UnitCounter` z listingu 4.1 zdefiniowaliśmy początkowe wartości obu zmiennych składowych:

```
var $units = 0;
var $weightPerUnit = 1.0;
```

Słowo kluczowe `var` jest wymagane po to, aby zaznaczyć, że `$units` i `$weightPerUnit` to zmienne składowe klasy. Gdy tworzony jest nowy obiekt `UnitCounter`, jego zmienne `$units` i `$weightPerUnit` otrzymują wartość odpowiednio `0` i `1.0`. Jeżeli w definicji klasy nie określono domyślnej wartości zmiennej, wówczas nie otrzymuje ona żadnej wartości.

Jawne deklarowanie zmiennych składowych w taki sposób, w jaki zrobiliśmy to na listingu 4.1, nie jest obowiązkowe. Jednak zalecamy właśnie takie postępowanie — deklarowanie zmiennych i nadawanie im wartości początkowych za każdym razem, ponieważ dzięki temu inni programiści, którzy będą wykorzystywali dany kod, będą od razu znali ich stan.

Funkcje składowe

Funkcje składowe klas można deklarować zarówno w PHP 4, jak i w PHP 5.

Definicje funkcji składowych umieszcza się w definicjach klas. Klasa `UnitCounter` z listingu 4.1 ma dwie funkcje składowe — `add()` i `totalWeight()`.

Obie te funkcje mogą odwoływać się do zmiennych składowych obiektu, na rzecz którego zostały wywołane, za pośrednictwem specjalnej zmiennej `$this`. Mówimy, że jest ona specjalna, ponieważ PHP rezerwuje za jej pośrednictwem miejsce w pamięci, które czeka na pojawienie się obiektu, który dopiero zostanie utworzony. Gdy wywoływana jest któraś z funkcji składowych, wartością tej zmiennej staje się obiekt, na rzecz którego ta funkcja jest wywoływana. Przyjrzyj się implementacji funkcji składowej `add()` klasy `UnitCounter` z listingu 4.1:

```
// Dodaj $n do całkowitej liczby sztuk. Niech domyślną wartością $n będzie 1.
function add($n = 1)
{
$this->units = $this->units + $n;
}
```

Funkcja ta dodaje wartość parametru `$n` do zmiennej składowej `$this->units`. Jeżeli nie zostanie przekazany żaden parametr, `$n` otrzyma domyślną wartość `1`. Gdy funkcja `add()` zostanie w poniższym przykładzie wywołana na rzecz obiektu `$bottles`,

```
// Utwórz nowy obiekt UnitCounter
$bottles = new UnitCounter;

// Wywołaj funkcję add()
$bottles->add(3);
```

zmienna `$this` w funkcji `add()` stanie się synonimem `$bottles`.

Funkcja składowa `totalWeight()` również odwołuje się do zmiennych składowych za pomocą wyrażenia `$this`. Zwraca ona całkowitą masę produktów mnożąc przez siebie wartości zmiennych składowych `$this->units` i `$this->weightPerUnit`.

```
// Utwórz nowy obiekt UnitCounter
$bricks = new UnitCounter;

$bricks->add(15);

// Wyświetla 15 – 15 sztuk po 1 kg każda
print $bricks->totalWeight();
```

PHP 5 pozwala na umieszczanie wyników zwracanych przez funkcje składowe w łańcuchach. Trzeba w tym celu otoczyć je nawiasami klamrowymi, tak jak to pokazaliśmy poniżej. Pokazujemy też alternatywne rozwiązanie, które można stosować w PHP 4:

```
// To polecenie działa tylko w PHP 5
print "Całkowita masa = {$bottles->totalWeight()} kg";

// To polecenie działa zarówno w PHP 4, jak i w PHP 5
print "Całkowita masa = " . $bottles->totalWeight() . " kg";
```

Umieszczanie definicji klas w osobnych plikach

Umieszczając definicję z listingu 4.1 w osobnym pliku, dajmy na to *UnitCounter.inc*, zyskujemy możliwość dołączania jej do innych skryptów poprzez wywoływanie poleceń `include` i `require`. Na listingu 4.2 pokazaliśmy przykładową dyrektywę `require` dołączającą do bieżącego skryptu definicję klasy `UnitCounter`.

Listing 4.2. Wykorzystywanie klasy UnitCounter

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html401/loose.dtd">
<html>
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
   <title>Wykorzystywanie klasy UnitCounter</title>
</head>
<body>
<?php
require "UnitCounter.inc";

// Utwórz nowy obiekt UnitCounter
$bottles = new UnitCounter;

// Ustaw licznik na 2 tuziny butelek
$bottles->units = 24;
```

```
// Dodaj jedną butelkę
$bottles->add();

// Dodaj jeszcze trzy
$bottles->add(3);

// Pokaż całkowitą liczbę butelek i ich masę
print "Jest {$bottles->units} sztuk, ";
print "całkowita masa = " . $bottles->totalWeight() . " kg";

// Zmień domyślną masę jednej sztuki i pokaż nową masę całkowitą
$bottles->weightPerUnit = 1.2;
print "<br>Prawidłowa masa całkowita = " . $bottles->totalweight() . " kg";

?>
</body>
</html>
```

Dyrektywy `include` i `require` przedstawiliśmy już w rozdziale 2. Dalsze przykłady ich wykorzystania zamieściliśmy w rozdziałach 6. i 16., w których opracowujemy biblioteki dla naszego sklepu internetowego *Hugh i Dave*.

Konstruktory

PHP 4 pozwala na definiowanie konstruktorów tylko w jeden sposób, a PHP 5 na dwa sposoby.

Jak już wcześniej pisaliśmy, gdy tworzony jest obiekt klasy `UnitCounter` zdefiniowanej na listingu 4.1, PHP inicjalizuje jego zmienne składowe `$units` i `$weightPerUnit` wartościami `0` i `1.0`. Jeżeli trzeba zmienić masę pojedynczego przedmiotu, można to zrobić już po utworzeniu obiektu, przypisując ją bezpośrednio do zmiennej składowej. Na przykład:

```
// Utwórz nowy obiekt UnitCounter
$bottles = new UnitCounter;
// Ustaw rzeczywistą masę butelki
$bottles->weightPerUnit = 1.2;
```

Jednak lepszym rozwiązaniem jest utworzenie *funkcji-konstruktora*, która będzie dbała o to, aby obiekt został przed pierwszym użyciem doprowadzony do odpowiedniego stanu. Jeżeli zdefiniujemy konstruktor, nie będziemy musieli inicjalizować niczego ręcznie, ponieważ PHP wywoła go automatycznie w chwili tworzenia obiektu.

PHP 5 pozwala na zadeklarowanie konstruktora w ciele definicji klasy poprzez umieszczenie w niej funkcji składowej o nazwie `__construct()` (słowo *construct* poprzedzone jest dwoma znakami podkreślenia). Na listingu 4.3 prezentujemy zmodyfikowaną klasę `UnitCounter`, rozbudowaną o konstruktor, który automatycznie ustawia prawidłową masę jednostkową.

Listing 4.3. Definiowanie konstruktora dla klasy UnitCounter

```
<?php

class UnitCounter
{
   var $units;
   var $weightPerUnit;

   function add($n = 1)
   {
      $this->units = $this->units + $n;
   }
```

```
    function totalWeight()
    {
       return $this->units * $this->weightPerUnit;
    }

    // Funkcja-konstruktor inicjalizująca zmienne składowe
    function __construct($unitWeight = 1.0)
    {
       $this->weightPerUnit = $unitWeight;
       $this->units = 0;
    }
}

?>
```

Ta definicja klasy odpowiada definicji z listingu 4.1. Jednak początkowe wartości zmiennych `$units` i `$weightPerUnit` nie są już podawane w ich deklaracjach, lecz są im nadawane przez funkcję składową `__construct()`. Obiekty nowej klasy `UnitCounter` pokazanej na listingu 4.3 tworzy się w następujący sposób:

```
// Utwórz obiekt UnitCounter, dla którego każdy przedmiot waży 1.2 kg
// - tyle co pełna butelka wina.
$bottles = new UnitCounter(1.2);
```

W chwili tworzenia obiektu PHP automatycznie wywołuje funkcję `__construct()` z parametrami podanymi po nazwie klasy. Tak więc w tym przykładzie do metody `__construct()` przekazywana jest wartość `1.2`, która jest następnie przypisywana do zmiennej `$bottles->weightPerUnit`.

W dalszym ciągu możliwe jest tworzenie obiektów `UnitCounter` bez przekazywania wartości do konstruktora, ponieważ dla zmiennej `$unitWeight` została zdefiniowana wartość domyślna równa `1.0`.

W PHP 5 można też zadeklarować konstruktor w inny sposób, a mianowicie definiując funkcję o takiej samej nazwie jak nazwa klasy. W PHP 4 była to jedyna dostępna metoda. Oznacza to, że na przykład na listingu 4.3 funkcję `__construct()` możemy zastąpić następującą funkcją:

```
function UnitCounter($weightPerUnit = 1)
{
   $this->weightPerUnit = $weightPerUnit;
   $this->units = 0;
}
```

Stosowanie funkcji o nazwie `__construct()` ułatwia zarządzanie dużymi projektami, ponieważ pozwala na łatwe przenoszenie klas, zmienianie ich nazw i wielokrotne wykorzystywanie ich w hierarchii klas bez konieczności modyfikowania ich zawartości. Hierarchie klas omówimy sobie w rozdziale 14.

Destruktory

Destruktory są dostępne jedynie w PHP 5.

Jeżeli w danej klasie został zdefiniowany konstruktor, zostanie on wywołany w chwili, gdy będzie tworzony jakiś obiekt tej klasy. Na podobnej zasadzie działają *destruktory* — są one wywoływane, gdy obiekt jest usuwany z pamięci. Podobnie jak inne zmienne w PHP, obiekty są usuwane z chwilą gdy zostają poza zasięgiem lub gdy zostanie jawnie wywołana funkcja `unset()` (o zasięgu zmiennych pisaliśmy w rozdziale 2.).

Definiowanie destruktora polega na umieszczeniu w definicji klasy funkcji `__destruct()` (podobnie jak w przypadku konstruktorów, słowo kluczowe *destruct* poprzedzone jest dwoma podkreślnikami, a `__destruct()` to nazwa zastrzeżona, której nie można stosować dla żadnych innych funkcji). Funkcje `__destruct()` nie mogą pobierać żadnych argumentów (w przeciwieństwie do funkcji `__construct()`), jednak mają one dostęp do zmiennych składowych usuwanych obiektów. PHP wywołuje destruktor tuż przed usunięciem składowych z pamięci.

Destruktory przydają się wtedy, gdy przed usunięciem obiektu z pamięci chcemy wykonać jeszcze jakieś czynności porządkowe. Możemy, na przykład, chcieć w elegancki sposób zamknąć połączenie z SZBD lub zapisać ustawienia wprowadzone przez użytkownika do pliku. Podczas tworzenia aplikacji zorientowanych obiektowo destruktory przydają się też przy diagnozowaniu błędów. Na przykład dodanie funkcji `__destruct()` do klasy `UnitCounter` zdefiniowanej na listingu 4.3 pozwala zorientować się, kiedy następuje usuwanie obiektów:

```
// Funkcja-destruktor wywoływana tuż przed usunięciem
// obiektu UnitCounter z pamięci
function __destruct()
{
   print "UnitCounter poza zasięgiem. Sztuk: {$this->units}";
}
```

Inny przykład destruktora pokażemy w podrozdziale „Statyczne zmienne składowe".

Prywatne zmienne składowe

Prywatne zmienne składowe można stosować w PHP 5.

Każdy skrypt korzystający z obiektów klasy `UnitCounter` zdefiniowanej na listingu 4.3 może bezpośrednio odwoływać się do ich zmiennych składowych `$units` i `$weightPerUnit`, ponieważ w klasie `UnitCounter` nie zaimplementowaliśmy żadnych mechanizmów, które zabezpieczałyby nas przed przekazywaniem do obiektów nieprawidłowych wartości. Przyjrzyj się na przykład następującemu fragmentowi kodu, w którym nie dość, że liczba sztuk zostaje wyrażona ułamkiem, to jeszcze masa jednostkowa otrzymuje ujemną wartość:

```
// Utwórz nowy obiekt UnitCounter
$b = new UnitCounter;

// Ustaw pewne wartości
$b->units = 7.3;
$b->weightPerUnit = -5.5;

$b->add(10);

// Pokaż całkowitą liczbę sztuk i masę
print "Jest {$b->units} sztuk, ";
print "masa całkowita = {$b->totalWeight()} kg";
```

Ten kod powoduje wyświetlenie następującej informacji:

```
Jest 17.3 sztuk, masa całkowita = -95.15 kg
```

W PHP 5 znacznie lepszym rozwiązaniem od tego, które stosowaliśmy dotychczas, jest zadeklarowanie zmiennych składowych jako *prywatnych* i zdefiniowanie funkcji składowych, za pośrednictwem których będzie można się do nich odwoływać. Na listingu 4.4 zmienne `$units` i `$weightPerUnit` zostały zadeklarowane właśnie jako prywatne.

Listing 4.4. Prywatne zmienne składowe

```
<?php
class UnitCounter
{
   private $units = 0;
   private $weightPerUnit = 1.0;

   function numberOfUnits()
   {
      return $this->units;
   }

   function add($n = 1)
   {
      if (is_int($n) && $n > 0)
         $this->units = $this->units + $n;

   }

   function totalWeight()
   {
      return $this->units * $this->weightPerUnit;
   }

   function __construct($unitWeight)
   {
      $this->weightPerUnit = abs((float)$unitWeight);
      $this->units = 0;
   }
}
?>
```

Po utworzeniu obiektu klasy `UnitCounter` zdefiniowanej na listingu 4.4, do zmiennych składowych `$units` i `$weightPerUnit` można się dostać jedynie za pośrednictwem funkcji zdefiniowanych w ciele klasy. Próba odwołania się do nich w tradycyjny sposób powoduje wystąpienie błędu:

```
// Utwórz obiekt klasy UnitCounter zdefiniowanej na listingu 4.4
$b = new UnitCounter(1.1);

// Te polecenia powodują wystąpienie błędu
$b->units = 7.3;
$b->weightPerUnit = -5.5;
```

Funkcja składowa `numberOfUnits()` daje dostęp do zmiennej `$units`, a funkcja składowa `add()` została ulepszona w taki sposób, aby całkowita liczba sztuk mogła być powiększana jedynie o całkowite wartości dodatnie. Ulepszyliśmy też funkcję `__construct()`, tak aby gwarantowała ona, że do zmiennej `$weightPerUnit` zostanie przypisana wartość dodatnia.

Definiowanie funkcji składowych pozwalających kontrolować sposób, w jaki są wykorzystywane zmienne składowe jest dobrą praktyką programistyczną. Jednak stosowanie takich środków ostrożności na niewiele się przyda, jeżeli nie uczynimy zmiennych składowych prywatnymi. Po prostu do tak zwanych publicznych zmiennych składowych można się odwoływać bezpośrednio i w prosty sposób zmieniać ich wartość.

Prywatne funkcje składowe

Prywatne funkcje składowe można stosować w PHP 5.

Jako prywatne można też zadeklarować funkcje składowe. Pozwala to ukryć szczegóły implementacyjne klasy, a to z kolei umożliwia modyfikowanie klasy w sposób, który nie wymaga późniejszej zmiany skryptów, w których jest ona wykorzystywana. Na listingu 4.5 pokazaliśmy, w jaki sposób klasa `FreightCalculator` ukrywa wewnętrzne metody wykorzystywane przez publicznie dostępną funkcję składową `totalFreight()`. Funkcja ta oblicza koszty przewozu towaru posiłkując się dwiema prywatnymi metodami — `perCaseTotal()` i `perKgTotal()`.

Listing 4.5. Prywatne funkcje składowe

```
class FreightCalculator
{
   private $numberOfCases;
   private $totalWeight;

   function totalFreight()
   {
      return $this->perCaseTotal() + $this->perKgTotal();
   }

   private function perCaseTotal()
   {
      return $this->numberOfCases * 1.00;
   }

   private function perKgTotal()
   {
      return $this->totalWeight * 0.10;
   }

   function __construct($numberOfCases, $totalWeight)
   {
      $this->numberOfCases = $numberOfCases;
      $this->totalWeight = $totalWeight;
   }
}
```

Podobnie jak do prywatnych zmiennych składowych, do prywatnych funkcji składowych można się dostać jedynie z wnętrza klasy, w której je zdefiniowano. Poniższy fragment kodu powoduje wystąpienie błędu:

```
// Utwórz obiekt klasy FreightCalculator zdefiniowanej na listingu 4.5
$f = new FreightCalculator(10, 150);

// Te polecenia powodują wystąpienie błędu
print $f->perCaseTotal();
print $f->perKgTotal();

// To polecenie jest OK — wyświetla wartość 25
print $f->totalFreight();
```

Statyczne zmienne składowe

Statyczne zmienne składowe można stosować w PHP 5.

PHP pozwala na deklarowanie zmiennych i funkcji składowych jako *statycznych*. Służy do tego słowo kluczowe `static`. Jak widziałeś w dotychczasowych przykładach, każdy obiekt danej klasy ma swoją własną, niezależną kopię zmiennych składowych. Jednak w przypadku składowych

statycznych jest inaczej. Są one współdzielone przez wszystkie obiekty danej klasy. Dzięki nim możemy wykorzystywać we wszystkich obiektach klasy te same wartości, bez konieczności deklarowania zmiennej globalnej, która byłaby dostępna w całym skrypcie.

Na listingu 4.6 definiujemy klasę `Donation` (darowizna). W każdym jej obiekcie będziemy przechowywali nazwisko ofiarodawcy i informację o kwocie, jaką zdecydował się przekazać — odpowiednio w zmiennej `$name` i `$amount`. Klasa ta będzie też śledziła całkowitą ilość zebranych pieniędzy i całkowitą liczbę zebranych datków. Wartości te będą przechowywane w zmiennych `$totalDonated` i `$numberOfDonors`, które będą dostępne dla wszystkich obiektów klasy. Każdy z nich będzie mógł odczytywać i zmieniać ich wartości. Do statycznych zmiennych składowych odwołujemy się przez *referencję do klasy*, a nie przez operator `->`. Dlatego na listingu 4.6 zmienne statyczne `$totalDonated` i `$numberOfDonors` są poprzedzone nazwą klasy (`Donation::`).

Listing 4.6. Statyczne zmienne składowe

```
<?php
class Donation
{
   private $name;
   private $amount;

   static $totalDonated = 0;
   static $numberOfDonors = 0;

   function info()
   {
      $share = 100 * $this->amount / Donation::$totalDonated;
      return "{$this->name} podarował {$this->amount} ({$share}%)";
   }

   function __construct($nameOfDonor, $donation)
   {
      $this->name = $nameOfDonor;
      $this->amount = $donation;

      Donation::$totalDonated = Donation::$totalDonated + $donation;
      Donation::$numberOfDonors++;
   }

   function __destruct()
   {
      Donation::$totalDonated = Donation::$totalDonated - $donation;
      Donation::$numberOfDonors--;
   }

}
?>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html401/loose.dtd">
<html>
<head>
   <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-2">
   <title>Korzystanie z klasy Donation</title>
</head>
<body>
<pre>
```

```
<?php
   $donors = array(
      new Donation("Mikołaj", 85.00),
      new Donation("Bartłomiej", 50.00),
      new Donation("Tymoteusz", 90.00),
      new Donation("Grzegorz", 65.00));

   foreach ($donors as $donor)
      print $donor->info() . "\n";

   $total = Donation::$totalDonated;
   $count = Donation::$numberOfDonors;
   print "Suma darowizn = {$total}\n";
   print "Liczba ofiarodawców = {$count}\n";

?>
</pre>
</body>
</html>
```

Wartości zmiennych statycznych `$totalDonated` i `$numberOfDonors` są uaktualniane w funkcji `__construct()`. Wartość `$donation` jest dodawana do wartości `$totalDonated`, a zmienna `$numberOfDonors` jest inkrementowana. Zdefiniowaliśmy też funkcję `__destruct()`, która dekrementuje zmienne `$totalDonated` i `$numberOfDonors` w chwili, gdy jakiś obiekt klasy `Donation` jest usuwany z pamięci.

Kod pokazany na listingu 4.6 definiuje najpierw klasę `Donation`, a następnie tworzy tablicę obiektów i wyświetla informację o liczbie darowizn i o całkowitej zebranej kwocie:

```
$total = Donation::$totalDonated;
$count = Donation::$numberOfDonors;
print "Suma darowizn = {$total}\n";
print "Liczba ofiarodawców = {$count}\n";
```

Ten fragment kodu pokazuje, że do statycznych zmiennych składowych można się odwoływać również spoza klasy, o ile tylko poprzedzi się ich nazwy przedrostkiem `Donation::`. Do zmiennych tych nie odwołujemy się za pomocą operatora `->` (stosowanego wraz z nazwą obiektu), ponieważ nie są one związane z żadnym konkretnym obiektem.

W celu wyświetlenia informacji o wszystkich darowiznach zastosowaliśmy pętlę `foreach`, w której dla każdego obiektu `Donation` wywołujemy funkcję składową `info()`. Zwraca ona łańcuch zawierający nazwisko ofiarodawcy, informację o przekazanej przez niego kwocie i o tym, jak duży (procentowo) jest jego wkład. Ta ostatnia wartość jest obliczana poprzez podzielenie wartości przechowywanej w zmiennej `$this->amount` danego obiektu przez statyczną wartość `Donation::$totalDonated`.

Wyniki zwracane przez kod z listingu 4.6 wyglądają następująco:

```
Mikołaj podarował 85 (29.3103448276%)
Bartłomiej podarował 50 (17.2413793103%)
Tymoteusz podarował 90 (31.0344827586%)
Grzegorz podarował 65 (22.4137931034%)
Suma darowizn = 290
Liczba ofiarodawców = 4
```

W przeciwieństwie do innych zmiennych składowych, ze zmiennych statycznych można korzystać nawet wtedy, gdy nie ma jeszcze żadnego obiektu danej klasy. Jeżeli tylko skrypt ma dostęp do definicji klasy, możesz odwoływać się do zmiennych statycznych poprzedzając ich nazwy specyfikatorem klasy, tak jak w poniższym przykładzie:

```
// Uzyskaj dostęp do definicji klasy Donation
require "example.4-6.php";

// Teraz nadaj zmiennym statycznym żądane wartości
Donation::$totalDonated = 124;
Donation::$numberOfDonors = 5;
```

Statyczne funkcje składowe

Statyczne funkcje składowe można stosować w PHP 5.

Deklaruje się je z użyciem słowa kluczowego `static` i podobnie jak w przypadku statycznych zmiennych składowych, odwołuje się do nich nie poprzez konkretne obiekty, lecz przez całą klasę, której nazwę umieszcza się przed nazwą funkcji. Możemy zmodyfikować listing 4.6 w taki sposób, aby dostęp do statycznych zmiennych składowych mógł odbywać się za pośrednictwem statycznych funkcji składowych:

```
private static $totalDonated = 0;
private static $numberOfDonors = 0;

static function total()
{
   return Donation::$totalDonated;
}

static function numberOfDonors()
{
   return Donation::$numberOfDonors;
}
```

Kod wykorzystujący zmodyfikowaną wersję klasy `Donation` będzie teraz mógł uzyskiwać dostęp do wartości zmiennych `$totalDonated` i `$numberOfDonors` za pośrednictwem statycznych funkcji `Donation::total()` i `Donation::numberOfDonors()`.

Funkcje statyczne mogą operować jedynie na statycznych zmiennych składowych i nie mogą wykonywać żadnych operacji na obiektach. Właśnie dlatego w ich ciele nie może się nigdy znaleźć odwołanie do zmiennej `$this`.

Podobnie jak do statycznych zmiennych składowych, do statycznych funkcji składowych można się odwoływać bez konieczności tworzenia obiektów danej klasy. Oczywiście statyczne zmienne i funkcje składowe z listingu 4.6 mogliśmy zadeklarować jako zmienne globalne i zwykłe funkcje zdefiniowane przez użytkownika. Jednak zdefiniowanie ich jako statycznych ułatwia grupowanie elementów o zbliżonym działaniu w definicjach klas, co jest zgodne z modułowym podejściem do tworzenia kodu.

Klonowanie obiektów.

W PHP 4 obiekty są klonowane zawsze, a w PHP 5 jest to opcjonalne. W następnym podrozdziale wyjaśnimy o co chodzi.

Klonowanie w PHP 5

Gdy tworzony jest nowy obiekt, PHP 5 zwraca referencję do niego, a nie sam obiekt. Tak więc zmienna, do której przypisujemy obiekt, jest tak naprawdę referencją do obiektu. Jest to istotna różnica w stosunku do tego, co było w PHP 4, kiedy to do zmiennych były przypisywane

fizyczne obiekty. Utworzenie kopii zmiennej obiektowej w PHP 5 powoduje po prostu utworzenie drugiej referencji do tego samego obiektu. Można to zaobserwować na przykładzie poniższego fragmentu kodu, który tworzy nowy obiekt klasy `UnitCounter` zdefiniowanej na listingu 4.1:

```
// Utwórz obiekt UnitCounter
$a = new UnitCounter();

$a->add(5);
$b = $a;
$b->add(5);

// Wyświetli "Liczba sztuk = 10";
print "Liczba sztuk = {$a->units}";
```

Gdy chcemy utworzyć nową, niezależną kopię jakiegoś obiektu, musimy użyć metody `__clone()`. PHP 5 udostępnia domyślną postać tej metody, która tworzy nowy, identyczny obiekt i kopiuje wartości poszczególnych składowych. Weźmy pod uwagę poniższy fragment kodu:

```
// Utwórz obiekt UnitCounter
$a = new UnitCounter();

$a->add(5);
$b = $a->__clone();
$b->add(5);

// Wyświetli "Liczba sztuk = 5"
print "Liczba sztuk = {$a->units}";

// Wyświetli "Liczba sztuk = 10"
print "Liczba sztuk = {$b->units}";
```

Kod ten tworzy obiekt `$a` i dodaje do niego pięć sztuk towaru za pośrednictwem polecenia `$a->add(5)`. W rezultacie w obiekcie `$a` znajduje się dokładnie 5 sztuk towaru. Następnie obiekt `$a` jest klonowany, a klon jest przypisywany do nowego obiektu `$b`. Potem do nowego obiektu `$b` dodawane jest 5 sztuk towaru, w wyniku czego w obiekcie tym znajduje się ostatecznie 10 sztuk towaru. Po wyświetleniu liczby towarów przechowywanych w oryginalnym obiekcie `$a` okazuje się, że faktycznie jest ich 5, a w obiekcie `$b`, że jest ich 10.

Możemy wpływać na sposób kopiowania obiektów umieszczając w definicjach naszych klas własną funkcję `__clone()`.Gdybyśmy, na przykład, chcieli, aby sklonowany obiekt `UnitCounter` przechowywał w zmiennej `$weightPerUnit` taką samą wartość jak oryginał, ale żeby wartość zmiennej `$units` została wyzerowana, moglibyśmy umieścić w definicji klasy następującą funkcję:

```
function __clone()
{
   $this->weightPerUnit = $that->weightPerUnit;
   $this->units = 0;
}
```

Do oryginalnego, źródłowego obiektu odwołujemy się w funkcji `__clone()` poprzez specjalną zmienną `$that`, a zmienna `$this` służy do odwoływania się do nowego obiektu, czyli do klonu.

Klonowanie w PHP 4

PHP 4 nie stosuje domyślnie referencji. Wszystkie nowotworzone obiekty są bezpośrednio przypisywane do zmiennych. Gdy zmienna obiektowa jest kopiowana, PHP 4 automatycznie klonuje obiekt. Rozważmy na przykład następujący fragment kodu w PHP 4:

```
// Utwórz obiekt UnitCounter
$a = new UnitCounter();

$a->add(5);
$b = $a;
$b->add(5);

// Wyświetli "Liczba sztuk = 5"
print "Liczba sztuk = {$a->units}";

// Wyświetli " Liczba sztuk = 10"
print "Liczba sztuk = {$b->units}";
```

Zmienna `$b` jest klonem (kopią) zmiennej `$a`, w związku z czym modyfikowanie jej nie wpływa w żaden sposób na zmienną `$a`.

Jeżeli nie chcemy klonować obiektu, możemy użyć przypisania referencyjnego `=&`, dzięki któremu zostanie skopiowana sama referencja. W poniższym fragmencie kodu zmienna `$b` jest przypisywana do zmiennej `$a` jako referencja do obiektu `UnitCounter`:

```
// Utwórz obiekt UnitCounter
$a = new UnitCounter();

$a->add(5);
$b =& $a;
$b->add(5);

// Wyświetli " Liczba sztuk = 10"
print " Liczba sztuk = {$a->units}";

// Wyświetli " Liczba sztuk = 10"
print " Liczba sztuk = {$b->units}";
```

Referencje i operator przypisania referencyjnego `=&` omówiliśmy już w rozdziale 2.

Dziedziczenie

Dziedziczenie jest dostępne w PHP 4 i PHP 5.

Jednym ze wspaniałych rozwiązań dostępnych w programowaniu zorientowanym obiektowo jest *dziedziczenie*. Pozwala ono na definiowanie nowych klas poprzez rozszerzanie możliwości klas już istniejących, nazywanych w tym przypadku *klasami bazowymi* lub inaczej *podstawowymi*. W PHP tworzenie nowych klas poprzez rozszerzanie istniejących wymaga stosowania słowa kluczowego `extends`.

Na listingu 4.7 rozszerzamy klasę `UnitCounter` z listingu 4.4 tworząc nową klasę `CaseCounter`. Jej zadaniem będzie zliczanie opakowań potrzebnych do przechowywania przedmiotów, których liczbę nadzoruje klasa `UnitCounter`. Jeżeli, na przykład, tymi przedmiotami będą butelki wina, to każda skrzynka będzie mogła pomieścić ich 12.

Listing 4.7. Definiowanie klasy CaseCounter poprzez rozszerzanie klasy UnitCounter

```
<?php

// Dostęp do definicji klasy UnitCounter
require "example.4-4.php";

class CaseCounter extends UnitCounter
{
   var $unitsPerCase;
```

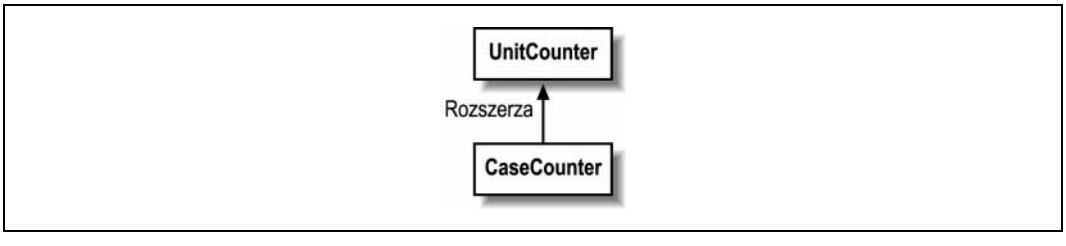
```
    function addCase()
    {
       $this->add($this->unitsPerCase);
    }

    function caseCount()
    {
       return ceil($this->units/$this->unitsPerCase);
    }

    function CaseCounter($caseCapacity)
    {
       $this->unitsPerCase = $caseCapacity;
    }
}

?>
```

Zanim omówimy implementację klasy `CaseCounter`, powinniśmy przyjrzeć się jej związkowi z klasą `UnitCounter`. Na rysunku 4.1 został on przedstawiony w postaci prostego *diagramu klas*. Tego rodzaju diagramy można tworzyć na bardzo wiele sposobów. My zdecydowaliśmy się pokazać relację dziedziczenia łącząc dwie klasy strzałką.



Rysunek 4.1. Diagram klas ukazujący zależność między klasami UnitCounter i CaseCounter

Nowa klasa `CaseCounter` oferuje możliwość zliczania opakowań, w których znajduje się pewna liczba przedmiotów jednego typu — na przykład butelek wina. Natomiast klasa bazowa `UnitCounter` oferuje możliwość zliczania przedmiotów i wyliczania ich całkowitej masy.

Aby utworzyć obiekt `CaseCounter` musimy określić, ile przedmiotów można przechowywać w jednym opakowaniu. Wartość ta zostanie przekazana do konstruktora tworzonego obiektu:

```
// Utwórz obiekt CaseCounter umieszczający po 12 butelek w skrzynce
$order = new CaseCounter(12);
```

a następnie zapamiętana w zmiennej składowej `$unitsPerCase`.

Funkcja składowa `addCase()` wykorzystuje przy zliczaniu opakowań zmienną składową `$unitsPerCase`:

```
function addCase()
{
    // Funkcja add() została już zdefiniowana
    // w klasie bazowej UnitCounter
    $this->add($this->unitsPerCase);
}
```

Dodawanie poszczególnych przedmiotów odbywa się poprzez wywoływanie funkcji składowej `add()` klasy bazowej `UnitCounter`. Wszystkie zmienne i funkcje składowe, które nie zostały zadeklarowane w klasie bazowej jako prywatne, mogą być wywoływane w klasach potomnych. Można się do nich odwoływać za pomocą operatora `->` i specjalnej zmiennej `$this`.

Funkcja składowa `caseCount()` oblicza, ile opakowań potrzeba do zapakowania całkowitej liczby przedmiotów. Jeżeli na przykład mamy 50 butelek wina, a w każdej skrzynce mieści się 12, to potrzebujemy 5 skrzynek. Liczba skrzynek jest więc ustalana poprzez podzielenie całkowitej liczby przedmiotów (przechowywanej w zmiennej składowej `$units` zadeklarowanej w klasie `UnitCounter`) przez wartość przechowywaną w zmiennej składowej `$unitsPerCase`. Wynik tej operacji jest zaokrąglany w górę do pełnej skrzynki za pomocą funkcji `ceil()`. Opisywaliśmy ją w rozdziale 3.

Po utworzeniu nowego obiektu `CaseCounter` będzie można się w nim odwoływać do wszystkich publicznie dostępnych zmiennych i funkcji klasy bazowej. Oznacza to, że możemy korzystać z obiektu `CaseCounter` dokładnie tak samo, jak gdyby był to obiekt `UnitCounter`, tyle że rozbudowany o nowe możliwości wprowadzone w klasie `CaseCounter`. Przyjrzyjmy się następującemu przykładowi:

```
// Utwórz obiekt CaseCounter przechowujący po 12 butelek w skrzynce
$order = new CaseCounter(12);

// Dodaj siedem butelek korzystając z funkcji zdefiniowanej w klasie UnitCounter
$order->add(7);
// Dodaj skrzynkę korzystając z funkcji zdefiniowanej w klasie CaseCounter
$order->addCase();

// Wyświetl całkowitą liczbę przedmiotów : 19
print $order->units;

// Wyświetl całkowitą liczbę skrzyń: 2
print $order->caseCount();
```

W przeciwieństwie do niektórych innych języków programowania, PHP pozwala definiować nowe klasy tylko na podstawie jednej klasy bazowej. Dziedziczenie z wielu klas bazowych mogłoby tylko niepotrzebnie skomplikować kod, a poza tym nie jest ono zbyt przydatne w praktyce. W rozdziale 14. opiszemy zaawansowane techniki programistyczne, które eliminują potrzebę stosowania dziedziczenia tego typu.

Wywoływanie konstruktorów klas podstawowych

Możliwość wywoływania konstruktorów klas podstawowych jest dostępna w PHP 5.

Obiekty `CaseCounter` wykorzystują trzy zmienne składowe. Dwie z nich są zdefiniowane w klasie `UnitCounter`, a jedna w klasie `CaseCounter`. Gdy tworzony jest obiekt klasy `CaseCounter`, PHP wywołuje funkcję `__construct()` zdefiniowaną w klasie `CaseCounter` i przypisuje zmiennej składowej `$unitsPerCase` wartość przekazaną w parametrze. W poniższym fragmencie kodu do funkcji `__construct()` przekazywana jest wartość `12`:

```
// Utwórz obiekt CaseCounter przechowujący po 12 butelek w skrzynce
$order = new CaseCounter(12);
```

PHP wywołuje jedynie funkcję `__construct()` zdefiniowaną w klasie `CaseCounter`. Konstruktor klasy bazowej `UnitCounter` nie jest automatycznie wywoływany. Dlatego też obiekty klasy `CaseCounter` z listingu 4.7 mają zaraz po utworzeniu zadeklarowaną masę przedmiotu równą 1 kg. Właśnie taką wartość ma bowiem zmienna składowa klasy bazowej. W klasie `CaseCounter` pokazanej na listingu 4.8 problem ten rozwiązano definiując funkcję `__construct()`, która wywołuje konstruktor klasy `UnitCounter`, odwołując się do niej za pośrednictwem wyrażenia `parent::`.

Listing 4.8. Wywoływanie konstruktora klasy bazowej

```
<?php

// Uzyskaj dostęp do definicji klasy UnitCounter
include "example.4-4.php";

class CaseCounter extends UnitCounter
{
   private $unitsPerCase;

   function addCase()
   {
      $this->add($this->unitsPerCase);
   }
   function caseCount()
   {
      return ceil($this->units/$this->unitsPerCase);
   }

   function __construct($caseCapacity, $unitWeight)
   {
      parent::__construct($unitWeight);
      $this->unitsPerCase = $caseCapacity;
   }
}

?>
```

Kod przedstawiony na listingu 4.8 wykorzystuje możliwości, jakie daje PHP 5. Rozszerzamy w nim bardziej zaawansowaną wersję klasy `UnitCounter`, z listingu 4.4. Zmienna składowa `$unitsPerCase` jest teraz zadeklarowana jako prywatna. Poza tym wykorzystujemy funkcję `__construct()` dostępną w PHP 5. Funkcja-konstruktor ulepszonej klasy `CaseCounter` pokazanej na listingu 4.8 pobiera teraz także drugi parametr, `$unit_Weight`, który jest przekazywany do funkcji `__construct()` zdefiniowanej w klasie `UnitCounter`.

Redefiniowanie funkcji

Redefiniowanie funkcji można stosować zarówno w PHP 4, jak i w PHP 5, a w PHP 5 dodatkowo jest dostępny operator `parent::` i operatory referencji do klas.

Redefiniowanie funkcji polega na ponownym zdefiniowaniu w *klasie pochodnej* funkcji dostępnych w klasie bazowej. Po utworzeniu obiektów klasy pochodnej redefiniowane funkcje mają w nich wyższy priorytet niż funkcje zdefiniowane w klasie bazowej. Z redefiniowaniem funkcji spotkaliśmy się już na listingu 4.8, na którym funkcja `__construct()` klasy bazowej `UnitCounter` została ponownie zdefiniowana w klasie `CaseCounter`.

Przyjrzyjmy się klasom `Shape` (kształt) i `Polygon` (wielokąt) zdefiniowanym w poniższym fragmencie kodu:

```
class Shape
{
   function info()
   {
      return "Shape.";
   }
}
```

```
class Polygon extends Shape
{
   function info()
   {
      return "Polygon.";
   }
}
```

Klasa `Shape` jest klasą bazową klasy `Polygon`, a tym samym klasa `Polygon` jest potomkiem klasy `Shape`. W obu klasach zdefiniowana jest funkcja `info()`. Zgodnie z podaną wcześniej regułą, gdy zostanie utworzony obiekt klasy `Polygon`, wszystkie wywołania funkcji `info()` będą dotyczyły funkcji zdefiniowanej w klasie `Polygon`.

Można się o tym przekonać wykonując następujący fragment kodu:

```
$a = new Shape;
$b = new Polygon;

// Wyświetla "Shape."
print $a->info();

// Wyświetla "Polygon."
print $b->info();
```

PHP 5 daje nam możliwość odwoływania się do funkcji `info()` z klasy bazowej — musimy poprzedzić jej nazwę wyrażeniem `parent::`. Możemy na przykład zmodyfikować definicję funkcji `info()` klasy `Polygon` w następujący sposób:

```
class Polygon extends Shape
{
   function info()
   {
      return parent::info() . "Polygon.";
   }
}

$b = new Polygon;

// Wyświetla "Shape.Polygon."
print $b->info();
```

Takie rozwiązanie można stosować też w klasach pochodnych. Dzięki niemu funkcje klas pochodnych zyskują cechy wszystkich funkcji o takiej samej nazwie zdefiniowanych w klasach nadrzędnych. Weźmy pod uwagę klasę `Triangle` (trójkąt), która rozszerza klasę `Polygon`:

```
class Triangle extends Polygon
{
   function info()
   {
      return parent::info() . "Triangle.";
   }
}

$t = new Triangle;

// Wyświetl "Shape.Polygon.Triangle."
print $t->info();
```

Za pomocą operatora `parent::` możemy dostać się jedynie do funkcji zdefiniowanej w klasie, z której bezpośrednio dziedziczymy. PHP daje też możliwość korzystania z funkcji zdefiniowanych w dowolnej klasie nadrzędnej naszej klasy — za pośrednictwem operatora referencji do klasy, który poznałeś już przy okazji omawiania statycznych zmiennych i funkcji składowych

w podrozdziale „Klasy i obiekty”. Możemy przepisać klasę `Triangle` w taki sposób, aby wywoływała ona bezpośrednio funkcję `info()` zdefiniowaną w jej przodku (to znaczy w klasie nadrzędnej):

```
class Triangle extends Polygon
{
   function info()
   {
      return Shape::info() . Polygon::info() . "Triangle.";
   }
}

$t = new Triangle;

// Wyświetla "Shape.Polygon.Triangle."
print $t->info();
```

Stosowanie operatorów dostępu do klasy czyni kod mniej przenośnym. Jeżeli, na przykład, podjęlibyśmy decyzję, że klasa `Triangle` powinna jednak być bezpośrednim potomkiem klasy `Shape`, musielibyśmy zmodyfikować jej implementację. Zastosowanie operatora referencyjnego `parent::` pozwala na łatwiejsze modyfikowanie hierarchii klas.

Składowe chronione

Składowe chronione można stosować w PHP 5.

W definicjach zmiennych i funkcji składowych może się pojawić słowo kluczowe `protected`, które uczyni z nich składowe *chronione*. Jest to rozwiązanie pośrednie między składowymi publicznymi, a prywatnymi. Daje ono klasie dostęp do zmiennych i funkcji zdefiniowanych w klasach nadrzędnych, ale nie pozwala na odwoływanie się do nich w klasach należących do innej hierarchii klas. Oznacza to, że na przykład klasa potomna ma dostęp do chronionych funkcji klasy nadrzędnej, ale nie są one dostępne dla klas, które nie mają z nią żadnego związku, ani dla innych elementów skryptu.

Na listingu 4.5 pojawiła się po raz pierwszy klasa `FreightCalculator`, której zadaniem jest obliczanie kosztów przewozu i całkowitej masy określonej liczby skrzynek. Klasa ta oblicza te wartości korzystając z dwóch prywatnych funkcji — `perCaseTotal()` i `perKgTotal()`.

Na listingu 4.9 przepisaliśmy definicję tej klasy, tym razem czyniąc te funkcje chronionymi. Pozwoli nam to wyprowadzić z klasy `FreightCalculator` nową klasę `AirFreightCalculator` (odnoszącą się do przewozów drogą lotniczą) i dokonać redefinicji funkcji, co pozwoli nam na stosowanie innych stawek za kilogram i za każdą skrzynkę.

Listing 4.9. Kalkulator kosztów przewozu towarów drogą lotniczą

```
class FreightCalculator
{
   protected $numberOfCases;
   protected $totalWeight;

   function totalFreight()
   {
      return $this->perCaseTotal() + $this->perKgTotal();
   }
```

```
    protected function perCaseTotal()
    {
       return $this->numberOfCases * 1.00;
    }

    protected function perKgTotal()
    {
       return $this->totalWeight * 0.10;
    }

    function __construct($numberOfCases, $totalWeight)
    {
       $this->numberOfCases = $numberOfCases;
       $this->totalWeight = $totalWeight;
    }
}

class AirFreightCalculator extends FreightCalculator
{

    protected function perCaseTotal()
    {
       // 15 zł + 1 zł za każde opakowanie
       return 15 + $this->numberOfCases * 1.00;
    }

    protected function perKgTotal()
    {
       // 0.40 zł za kilogram
       return $this->totalWeight * 0.40;
    }
}
```

Ponieważ implementacja funkcji `perCaseTotal()` i `perKgTotal()` w klasie `AirFreightCalculator` wymaga dostępu do zmiennych składowych `$totalWeight` i `$numberOfCases` klasy `FreightCalculator`, zostały one zadeklarowane jako chronione.

Metody finalne

Deklarowanie metod jako finalnych możliwe jest w PHP 5.

W klasie `AirFreightCalculator` z listingu 4.9 nie pojawia się nowa definicja funkcji składowej `totalFreight()`, ponieważ nie ma takiej potrzeby. Jej wersja zdefiniowana w klasie `FreightCalculator` prawidłowo oblicza opłatę. Istnieje możliwość zabezpieczenia się przed sytuacją, w której któraś z metod klasy bazowej mogłaby zostać ponownie zdefiniowana w klasie potomnej. Wystarczy zadeklarować taką metodę jako *finalną*. Umieszczenie w definicji funkcji składowej `totalFreight()` słowa kluczowego `final` zabezpiecza przed przypadkowym ponownym zdefiniowaniem tej funkcji:

```
final function totalFreight()
{
   return $this->perCaseTotal() + $this->perKgTotal();
}
```

Zgłaszanie i obsługiwanie wyjątków

W PHP 5 pojawił się model obsługi wyjątków pozwalający na opisywanie błędów za pośrednictwem specjalnych obiektów. Mogą one pojawiać się w programie i zostać „wychwycone" przez specjalne procedury obsługi błędów. Do zgłaszania wyjątków służy wyrażenie `throw`, a do ich wychwytywania `try...catch`.

Dzięki wyrażeniom `try...catch` w sytuacjach awaryjnych program może przeskoczyć bezpośrednio do kodu obsługi danego błędu. Zamiast kończyć wykonywanie skryptu podaniem informacji o zaistnieniu błędu krytycznego, PHP *zgłasza* wyjątki, które mogą zostać w programie *wychwycone* i przetworzone. Wyrażenie `throw` jest zawsze stosowane w połączeniu z wyrażeniem `try...catch`. W najprostszym przypadku wygląda to tak, jak pokazujemy poniżej:

```
$total = 100;
$n = 5;

$result;

try
{
   // Sprawdź wartość $n zanim jej użyjesz
   if ($n == 0)
      throw new Exception("Nie mogę przypisać n wartości zero! ");

   // Oblicz średnią
   $result = $total / $n;
}
catch (Exception $x)
{
   print "Wystąpił błąd: {$x->getMessage()};
}
```

Blok poleceń umieszczony w nawiasach klamrowych po słowie kluczowym `try` jest wykonywany jak standardowa część programu. Nawiasy są obowiązkowe — nawet jeśli miałoby się w nich znaleźć tylko jedno polecenie. Jeżeli w bloku `try` wywołane zostanie wyrażenie `throw`, wówczas zostaną wykonane wszystkie polecenia umieszczone w nawiasach klamrowych po słowie kluczowym `catch`. Wyrażenie `throw` umieszcza w programie obiekt opisujący błąd, a wyrażenie `catch` „wychwytuje" go i przypisuje go do podanej zmiennej.

W wyrażeniu `catch` określa się też, jakiego rodzaju obiekty ma ono wychwytywać, umieszczając przed nazwą zmiennej nazwę ich klasy. Poniższy fragment kodu wychwytuje obiekty klasy `Exception` i przypisuje je do zmiennej `$x`:

```
catch (Exception $x)
{
   print "Wystąpił błąd: {$x->getMessage()};
}
```

Określanie w bloku `catch` rodzaju obiektu, jaki ma zostać wyłapany, jest przykładem podpowiedzi odnośnie rodzaju klasy. Podpowiedzi tego typu omawiamy w rozdziale 14.

Klasa Exception

Choć błędy można zgłaszać za pomocą dowolnych klas, w PHP 5 przewidziano predefiniowaną klasę `Exception` stworzoną specjalnie do tego celu. Jest ona tak skonstruowana, że świetnie się do tego nadaje.

Tworząc obiekt klasy `Exception` musimy określić dla niego treść komunikatu o błędzie i opcjonalny kod błędu będący liczbą całkowitą. Informacje te będzie można później odczytać za pomocą funkcji składowych `getMessage()` i `getCode()`. W każdym obiekcie `Exception` zapamiętywaną jest też nazwa skryptu i wiersz, w którym wystąpił błąd. Dostęp do tych informacji zapewniają funkcje składowe `getFile()` i `getLine()`. Wykorzystujemy je w kodzie pokazanym na listingu 4.10, w którym definiujemy funkcję `formatException()`, zwracającą dla podanego obiektu `$e` klasy `Exception` prosty komunikat o zaistniałym błędzie.

Listing 4.10. Prosta sekwencja try-catch

```
<?php

function formatException(Exception $e)
{
   return "Błąd {$e->getCode()}: {$e->getMessage()}
      (linia: {$e->getline()} w pliku {$e->getfile()})";
}

function average($total, $n)
{
   if ($n == 0)
      throw new Exception("Liczba sztuk = 0", 1001);

   return $total / $n;
}

// Skrypt, w którym wykorzystuje się funkcję average()
try
{
   $a = average(100, 0);
   print "Średnia = {$a}";
}
catch (Exception $error)
{
   print formatException($error);
}

?>
```

Na listingu 4.10 pokazaliśmy jak za pomocą wyrażenia `try...catch` należy wychwytywać wyjątki zgłaszane przez funkcję `average()`. Tworzony jest obiekt `Exception` z odpowiednim komunikatem i kodem błędu, który jest zgłaszany przez funkcję `average()` w sytuacji, gdy zmienna `$n` ma wartość zero. Na listingu 4.10 funkcja `average()` wywoływana jest w bloku `try`. Dzięki temu, jeżeli zgłosi ona wyjątek, zostanie on wychwycony przez blok `catch` i zostanie wywołana funkcja `formatException()`. Nada ona komunikatowi przechowywanemu w obiekcie `$error` klasy `Exception` postać nadającą się do wyświetlenia na ekranie.

Po uruchomieniu kodu pokazanego na listingu 4.10 wywołanie funkcji `average()` powoduje zgłoszenie obiektu klasy `Exception`, co w rezultacie powoduje wyświetlenie następującego komunikatu:

```
Błąd 1001: Liczba sztuk = 0
   (linia: 13 w pliku c:\htdocs\book\example.4-10.php)
```

Gdyby funkcja `average()` z listingu 4.10 została wywołana poza blokiem `try...catch`, zgłoszony wyjątek nie zostałby wychwycony i nastąpiłoby zakończenie wykonywania skryptu z wyświetleniem komunikatu o nieobsłużonym wyjątku (ang. *uncaught exception*).

Wyrażenia `try...catch` stanowią alternatywę dla kończenia wykonywania skryptów poleceniami `exit()` i `die()`. Stosując je można tworzyć aplikacje, w których sytuacje awaryjne będą obsługiwane w przewidywalny sposób. Należy jednak pamiętać, że wyjątki dość mocno różnią się od ostrzeżeń i błędów generowanych przez PHP, gdy nie wszystko idzie tak jak trzeba. Wyrażenia `try...catch` nie pozwalają niestety na obsługę błędów krytycznych, takich jak na przykład dzielenie przez zero (możesz jednak uniknąć wyświetlania komunikatów o tego rodzaju błędach stosując operator `@`, który opiszemy w rozdziale 6.). Kod pokazany na listingu 4.10 implementuje funkcję `average()` w taki sposób, że przed wykonaniem dzielenia sprawdza ona wartość zmiennej `$n`. Pozwala to uniknąć wystąpienia krytycznego błędu dzielenia przez zero (ang. *division by zero*).

Zarządzaniem błędami i ostrzeżeniami zgłaszanymi przez PHP zajmiemy się w rozdziale 12.