

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

PHP5. Tajniki programowania

Autorzy: Andi Gutmans,
Stig Saether Bakken, Derick Rethans
Tłumaczenie: Daniel Kaczmarek (przedmowa,
rozdz. 1, 8-11), Radosław Meryk (rozdz. 12-16,
dod. A-C), Anna Zawadzka (rozdz. 2-7)
ISBN: 83-7361-856-2

Tytuł oryginału: [PHP 5 Power Programming](#)

Format: B5, stron: 728



Odkryj potęgę najnowszej wersji języka PHP

- Zaawansowane techniki programowania w PHP5
- Wzorce projektowe i stosowanie technologii XML i SOAP
- Sposoby poprawy wydajności aplikacji
- Współpraca z bazami danych

PHP w ciągu ostatnich lat stał się jednym z najpopularniejszych języków programowania wykorzystywanych do tworzenia aplikacji internetowych. Swoją sukces zawdzięcza prostocie i ogromnym możliwościom, pozwalającym na pisanie rozbudowanych aplikacji, znacznie przekraczających funkcjonalnością „zwykłe” portale i dynamiczne strony WWW. Najnowsza wersja języka PHP, oznaczona numerem 5, to w pełni obiektowy język programowania umożliwiający realizację złożonych projektów. Posiada mechanizmy obsługi plików XML i protokołu SOAP oraz poprawione i rozbudowane funkcje do komunikacji z bazami danych.

„PHP5. Tajniki programowania” to napisany przez współtwórcę języka PHP5 oraz dwóch doskonałych programistów przewodnik opisujący ogromne możliwości tej platformy. Autorzy przedstawiają sposoby wykorzystania PHP5 w projektach informatycznych o dowolnej skali złożoności. Omawiają model obiektowy PHP5, wzorce projektowe, metody korzystania z plików XML i technologii SOAP oraz techniki współpracy z bazami danych. W książce znajdziesz także szczegółowe omówienie biblioteki PEAR, obsługi wyjątków oraz metod optymalizowania wydajności aplikacji.

- Nowe możliwości PHP5
- Podstawy PHP5 i programowania zorientowanego obiektowo
- Stosowanie wzorców projektowych
- Techniki tworzenia aplikacji WWW
- Komunikacja z bazami danych, współpraca z MySQL i SQLite
- Obsługa błędów i wyjątków
- Przetwarzanie plików XML
- Instalowanie biblioteki PEAR
- Pakiety PEAR
- Tworzenie komponentów PEAR
- Przenoszenie kodu z PHP4 do wersji PHP5
- Projektowanie wydajnych aplikacji

Jeśli tworzysz aplikacje WWW, pakiety lub rozszerzenia PHP, w tej książce znajdziesz odpowiedzi na wszystkie pytania.

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



Spis treści

Słowo wstępne	15
Przedmowa	17
Rozdział 1. Co nowego w PHP5?	25
1.1. Wprowadzenie	25
1.2. Własności języka	25
1.2.1. Nowy model zorientowany obiektowo	25
1.2.2. Nowe mechanizmy zorientowane obiektowo	27
1.2.3. Pozostałe nowe mechanizmy języka	31
1.3. Ogólne zmiany w PHP	32
1.3.1. XML i usługi sieciowe	32
1.4. Pozostałe nowe mechanizmy PHP5	36
1.4.1. Nowy menedżer pamięci	36
1.4.2. Wycofana obsługa Windows 95	36
1.5. Podsumowanie	36
Rozdział 2. Podstawy języka PHP5	37
2.1. Wprowadzenie	37
2.2. Osadzanie w HTML	38
2.3. Komentarze	38
2.4. Zmienne	39
2.4.1. Pośrednie referencje do zmiennych	40
2.4.2. Zarządzanie zmiennymi	40
2.4.3. Superglobale	42
2.5. Podstawowe typy danych	43
2.5.1. Liczby całkowite	43
2.5.2. Liczby zmiennopozycyjne	43
2.5.3. Łańcuchy	44
2.5.4. Typ logiczny (boolowski)	46
2.5.5. Null	47
2.5.6. Identyfikator zasobów (resource)	48
2.5.7. Tablice	48
2.5.8. Stałe	54
2.6. Operatory	55
2.6.1. Operatory binarne	56
2.6.2. Operatory przypisania	56
2.6.3. Operatory porównania	57
2.6.4. Operatory logiczne	58
2.6.5. Operatory bitowe	59

2.6.6. Operatory jednoargumentowe	59
2.6.7. Operatory negacji	59
2.6.8. Operatory inkrementacji i dekrementacji	60
2.6.9. Operatory rzutowania	61
2.6.10. Operator kontroli błędów	61
2.6.11. Jedyne operator trójargumentowy (tzw. ternariusz)	62
2.7. Struktury kontrolne	62
2.7.1. Warunkowe struktury kontrolne	62
2.7.2. Struktury kontrolne w postaci pętli	65
2.7.3. Struktury kontrolne dołączania kodu	68
2.8. Funkcje	70
2.8.1. Funkcje definiowane przez użytkownika	71
2.8.2. Zasięg funkcji	71
2.8.3. Zwrocenie wartości przez wartość	72
2.8.4. Zwrocenie wartości przez referencję	73
2.8.5. Definiowanie parametrów funkcji	73
2.8.6. Zmienne statyczne	75
2.9. Podsumowanie	75
Rozdział 3. Język obiektowy	77
3.1. Wprowadzenie	77
3.2. Obiekty	78
3.3. Deklarowanie klasy	79
3.4. Słowo kluczowe new i konstruktory	79
3.5. Destruktory	80
3.6. Odwoływanie się do metod i właściwości przy użyciu zmiennej \$this	81
3.6.1. Właściwości public, protected i private	81
3.6.2. Metody public, protected i private	83
3.6.3. Właściwości statyczne	84
3.6.4. Metody statyczne	86
3.7. Stałe klasy	86
3.8. Klonowanie obiektów	87
3.9. Polimorfizm	89
3.10. parent:: i self::	91
3.11. Operator instanceof	92
3.12. Metody i klasy abstrakcyjne	93
3.13. Interfejsy	94
3.14. Dziedziczenie interfejsów	96
3.15. Metody final	96
3.16. Klasy final	97
3.17. Metoda __toString()	97
3.18. Obsługa wyjątków	98
3.19. __autoload()	101
3.20. Wskazania typu klasy w parametrach funkcji	103
3.21. Podsumowanie	104
Rozdział 4. Zaawansowane programowanie obiektowe i wzorce projektowe	105
4.1. Wprowadzenie	105
4.2. Możliwości przeciążania	105
4.2.1. Przeciążanie właściwości i metod	106
4.2.2. Przeciążanie składni dostępu do tablic	108
4.3. Iteratory	109
4.4. Wzorce projektowe	113
4.4.1. Wzorzec strategii	114
4.4.2. Wzorzec Singleton	116

4.4.3. Wzorzec fabryki	117
4.4.4. Wzorzec obserwatora	120
4.5. Refleksja (ang. reflection)	122
4.5.1. Wprowadzenie	122
4.5.2. Interfejs Reflector	123
4.5.3. Przykłady użycia refleksji	125
4.5.4. Implementowanie wzorca delegata przy użyciu refleksji	126
4.6. Podsumowanie	128
Rozdział 5. Jak napisać aplikację sieci WWW z PHP	129
5.1. Wprowadzenie	129
5.2. Osadzanie w kodzie HTML	130
5.3. Dane wprowadzane przez użytkownika	132
5.4. Zabezpieczanie danych wprowadzanych przez użytkownika	134
5.4.1. Pospolite błędy	135
5.5. Techniki zabezpieczania skryptów	137
5.5.1. Sprawdzanie danych wejściowych	137
5.5.2. Weryfikacja HMAC	139
5.5.3. PEAR::Crypt_HMAC	140
5.5.4. Program filtrujący	143
5.5.5. Praca z hasłami	144
5.5.6. Obsługa błędów	145
5.6. Cookies	146
5.7. Sesje	149
5.8. Wgrywanie plików (ang. upload)	153
5.8.1. Obsługiwanie przychodzącego wgrywanego pliku	153
5.9. Architektura	158
5.9.1. Jeden skrypt obsługuje wszystko	158
5.9.2. Jeden skrypt na funkcję	159
5.9.3. Oddzielanie logiki od układu	159
5.10. Podsumowanie	161
Rozdział 6. Bazy danych z PHP5	163
6.1. Wprowadzenie	163
6.2. MySQL	164
6.2.1. Mocne i słabe strony MySQL	164
6.2.2. Interfejs PHP	165
6.2.3. Przykładowe dane	166
6.2.4. Połączenia	166
6.2.5. Zapytania buforowane i niebuforowane	168
6.2.6. Zapytania	168
6.2.7. Zapytania z wieloma instrukcjami	169
6.2.8. Tryby pobierania	170
6.2.9. Zapytania preinterpretowane	170
6.2.10. Obsługa BLOB	173
6.3. SQLite	174
6.3.1. Mocne i słabe strony SQLite	174
6.3.2. Najlepsze obszary zastosowania	176
6.3.3. Interfejs PHP	176
6.4. PEAR DB	191
6.4.1. Uzyskiwanie PEAR BD	191
6.4.2. Wady i zalety abstrakcji baz danych	191
6.4.3. Które funkcje są abstrahowane	192
6.4.4. Połączenia z bazą danych	193

6.4.5. Wykonywanie zapytań	195
6.4.6. Pobieranie wyników	198
6.4.7. Sekwencje	200
6.4.8. Funkcje związane z przenośnością	201
6.4.9. Błędy abstrakcyjne	203
6.4.10. Metody złożone	205
6.5. Podsumowanie	207
Rozdział 7. Obsługa błędów	209
7.1. Wprowadzenie	209
7.2. Rodzaje błędów	210
7.2.1. Błędy programistyczne	210
7.2.2. Niezdefiniowane symbole	213
7.2.3. Błędy dotyczące przenośności	215
7.2.4. Błędy wykonania	219
7.2.5. Błędy PHP	219
7.3. Błędy PEAR	225
7.3.1. Klasa PEAR_Error	228
7.3.2. Obsługa błędów PEAR	231
7.3.3. Tryby błędów PEAR	232
7.3.4. Łagodna obsługa błędów	233
7.4. Wyjątki	235
7.4.1. Co to są wyjątki	235
7.4.2. try, catch i throw	236
7.5. Podsumowanie	238
Rozdział 8. XML i PHP5	239
8.1. Wprowadzenie	239
8.2. Słownictwo	240
8.3. Parsowanie XML-a	242
8.3.1. SAX	242
8.3.2. DOM	246
8.4. SimpleXML	250
8.4.1. Tworzenie obiektu SimpleXML	251
8.4.2. Przeglądanie obiektów SimpleXML	251
8.4.3. Zapisywanie obiektów SimpleXML	253
8.5. PEAR	253
8.5.1. XML_Tree	253
8.5.2. XML_RSS	254
8.6. Przekształcanie XML-a	257
8.6.1. XSLT	257
8.7. Komunikacja za pośrednictwem XML	261
8.7.1. XML-RPC	262
8.7.2. SOAP	269
8.8. Podsumowanie	275
Rozdział 9. Najważniejsze rozszerzenia	277
9.1. Wprowadzenie	277
9.2. Pliki i strumienie	278
9.2.1. Dostęp do plików	278
9.2.2. Dane wejściowe i wyjściowe z programu	280
9.2.3. Strumienie wejścia/wyjścia	283
9.2.4. Strumienie kompresji	284
9.2.5. Strumienie użytkownika	286
9.2.6. Strumienie URL	288

9.2.7. Blokowanie	292
9.2.8. Zmiana nazwy i usuwanie plików	293
9.2.9. Pliki tymczasowe	294
9.3. Wyrażenia regularne	295
9.3.1. Składnia	295
9.3.2. Funkcje	308
9.4. Obsługa dat	315
9.4.1. Odczytywanie daty i godziny	315
9.4.2. Formatowanie daty i godziny	318
9.4.3. Parsowanie dat w różnych formatach	326
9.5. Operacje na grafice przy użyciu GD	326
9.5.1. Przypadek 1: formularz odporny na działanie robotów	328
9.5.2. Przypadek 2: wykres słupkowy	333
9.5.3. Rozszerzenie Exif	338
9.6. Wielobajtowe ciągi i zestawy znaków	340
9.6.1. Konwersje zestawów znaków	341
9.6.2. Dodatkowe funkcje obsługujące zestawy znaków wielobajtowych	346
9.6.3. Ustawienia międzynarodowe	350
9.7. Podsumowanie	353

Rozdział 10. Stosowanie PEAR **355**

10.1. Wprowadzenie	355
10.2. Pojęcia związane z PEAR	356
10.2.1. Pakiety	356
10.2.2. Wersje	357
10.2.3. Numery wersji	358
10.3. Pobieranie repozytorium PEAR	360
10.3.1. Instalowanie dystrybucji PHP PEAR dla systemów UNIX/Linux	360
10.3.2. Windows	361
10.3.3. go-pear.org	362
10.4. Instalowanie pakietów	365
10.4.1. Polecenie pear	366
10.5. Parametry konfiguracyjne	369
10.6. Polecenia PEAR	376
10.6.1. Polecenie pear install	376
10.6.2. Polecenie pear list	380
10.6.3. Polecenie pear info	381
10.6.4. Polecenie pear list-all	382
10.6.5. Polecenie pear list-upgrades	382
10.6.6. Polecenie pear upgrade	383
10.6.7. Polecenie pear upgrade-all	384
10.6.8. Polecenie pear uninstall	385
10.6.9. Polecenie pear search	385
10.6.10. Polecenie pear remote-list	386
10.6.11. Polecenie pear remote-info	387
10.6.12. Polecenie pear download	387
10.6.13. Polecenie pear config-get	388
10.6.14. Polecenie pear config-set	388
10.6.15. Polecenie pear config-show	388
10.6.16. Skróty	389
10.7. Interfejsy instalatora	390
10.7.1. Instalator CLI (Command Line Interface)	390
10.7.2. Instalator Gtk	390
10.8. Podsumowanie	393

Rozdział 11. Najważniejsze pakiety PEAR	395
11.1. Wprowadzenie	395
11.2. Zapytania do bazy danych	395
11.3. Systemy szablonów	395
11.3.1. Pojęcia związane z szablonami	396
11.3.2. Szablon HTML_Template_IT	396
11.3.3. Szablon HTML_Template_Flexy	399
11.4. Uwierzytelnianie	404
11.4.1. Informacje ogólne	404
11.4.2. Przykład: Auth i plik z hasłami	405
11.4.3. Przykład: Auth i baza danych z danymi o użytkowniku	406
11.4.4. Bezpieczeństwo pracy z pakietem Auth	408
11.4.5. Skalowalność Auth	410
11.4.6. Podsumowanie informacji o Auth	411
11.5. Obsługa formularzy	411
11.5.1. Pakiet HTML_QuickForm	411
11.5.2. Przykład: formularz logowania	412
11.5.3. Pobieranie danych	412
11.6. Buforowanie	412
11.6.1. Pakiet Cache_Lite	413
11.7. Podsumowanie	414
Rozdział 12. Tworzenie komponentów PEAR	415
12.1. Wprowadzenie	415
12.2. Standardy PEAR	415
12.2.1. Nazewnictwo symboli	416
12.2.2. Wcięcia	418
12.3. Numery wersji	420
12.4. Środowisko wiersza polecenia	421
12.5. Podstawy	422
12.5.1. Kiedy i w jaki sposób można dołączać pliki	422
12.5.2. Obsługa błędów	423
12.6. Tworzenie pakietów	423
12.6.1. Przykład PEAR: HelloWorld	424
12.6.2. Tworzenie archiwum tarball	426
12.6.3. Weryfikacja	427
12.6.4. Testy regresji	428
12.7. Format pliku package.xml	428
12.7.1. Informacje o pakiecie	429
12.7.2. Informacje o wersji	431
12.8. Zależności	436
12.8.1. Element <deps>	436
12.8.2. Element <dep>	436
12.8.3. Typy zależności	437
12.8.4. Dlaczego należy unikać zależności	438
12.8.5. Zależności opcjonalne	439
12.8.6. Przykłady	439
12.9. Zastępowanie ciągów znaków	440
12.9.1. Element <replace>	440
12.9.2. Przykłady	440
12.10. Włączanie kodu w języku C	441
12.10.1. Element <configureoptions>	441
12.10.2. Element <configureoption>	441

12.11. Publikowanie pakietów	442
12.12. Proces publikowania pakietów PEAR	442
12.13. Przygotowanie pakietu	444
12.13.1. Analiza kodu źródłowego	444
12.13.2. Generowanie sum kontrolnych MD5	444
12.13.3. Aktualizacja pliku package.xml	444
12.13.4. Tworzenie archiwum tarball	445
12.14. Przesyłanie kodu	445
12.14.1. Przesyłanie wydania	445
12.14.2. Koniec	446
12.15. Podsumowanie	446
Rozdział 13. Migracja	447
13.1. Wprowadzenie	447
13.2. Model obiektowy	447
13.3. Przekazywanie obiektów do funkcji	448
13.4. Tryb zgodności	449
13.4.1. Konwersja obiektów	449
13.4.2. Porównywanie obiektów	450
13.5. Inne zmiany	451
13.5.1. Przypisywanie wartości do zmiennej \$this	451
13.5.2. get_class	454
13.6. E_STRICT	454
13.6.1. „Automagiczne” tworzenie obiektów	454
13.6.2. var i public	455
13.6.3. Konstruktory	455
13.6.4. Metody dziedziczone	456
13.6.5. Definiowanie klas przed ich wykorzystaniem	456
13.7. Inne problemy zgodności	456
13.7.1. Interfejs wiersza polecenia	457
13.7.2. Komentarze	457
13.7.3. MySQL	458
13.8. Zmiany w funkcjach	458
13.8.1. array_merge()	459
13.8.2. stripslashes() i stripslashes()	459
13.9. Podsumowanie	460
Rozdział 14. Wydajność	461
14.1. Wprowadzenie	461
14.2. Projektowanie wydajnych aplikacji	462
14.2.1. Wskazówka 1: uwaga na stany	462
14.2.2. Wskazówka 2: buforowanie	463
14.2.3. Wskazówka 3: nie należy przesadzać z projektowaniem	469
14.3. Testy szybkości	470
14.3.1. Zastosowanie programu ApacheBench	470
14.3.2. Zastosowanie programu Siege	471
14.3.3. Testowanie a rzeczywisty ruch	472
14.4. Tworzenie profilu aplikacji za pomocą programu Profiler z pakietu Zend Studio	472
14.5. Tworzenie profili za pomocą APD	473
14.5.1. Instalacja debugera APD	474
14.5.2. Analiza danych	475
14.6. Tworzenie profili za pomocą programu Xdebug	478
14.6.1. Instalacja programu Xdebug	478
14.6.2. Śledzenie wykonywania skryptu	479
14.6.3. Zastosowanie programu KCachegrind	481

14.7. Zastosowanie APC (Advanced PHP Cache)	483
14.8. Zastosowanie ZPS (Zend Performance Suite)	484
14.8.1. Automatyczna optymalizacja	484
14.8.2. Buforowanie skompilowanego kodu	485
14.8.3. Buforowanie dynamicznych stron	486
14.8.4. Kompresja stron	489
14.9. Optymalizacja kodu	490
14.9.1. Mikrotesty szybkości	491
14.9.2. Przepisywanie kodu w PHP na język C	493
14.9.3. Kod obiektowy a proceduralny	493
14.10. Podsumowanie	495
Rozdział 15. Wstęp do pisania rozszerzeń PHP	497
15.1. Wprowadzenie	497
15.2. Szybki start	498
15.2.1. Zarządzanie pamięcią	503
15.2.2. Zwracanie wartości z funkcji PHP	504
15.2.3. Dokończenie funkcji self_concat()	504
15.2.4. Podsumowanie przykładu	506
15.2.5. Tworzenie interfejsu dla bibliotek zewnętrznych	506
15.2.6. Zmienne globalne	515
15.2.7. Wprowadzanie własnych dyrektyw INI	515
15.2.8. Makra TSRM (Thread-Safe Resource Manager)	517
15.3. Podsumowanie	518
Rozdział 16. Tworzenie skryptów powłoki w PHP	521
16.1. Wprowadzenie	521
16.2. Wykorzystanie PHP do pisania skryptów powłoki działających w wierszu polecenia	521
16.2.1. Różnice pomiędzy wersjami CLI i CGI	522
16.2.2. Środowisko wykonywania skryptów powłoki	524
16.2.3. Przetwarzanie opcji wiersza polecenia	526
16.2.4. Dobre praktyki	529
16.2.5. Zarządzanie procesami	530
16.2.6. Przykłady	533
16.3. Podsumowanie	539
Dodatek A Zestawienie pakietów repozytoriów PEAR i PECL	541
A.1. Uwierzytelnianie	541
A.2. Testy szybkości	545
A.3. Buforowanie	545
A.4. Konfiguracja	546
A.5. Obsługa konsoli	547
A.6. Bazy danych	549
A.7. Godziny i daty	560
A.8. Szyfrowanie	562
A.9. Formaty plików	564
A.10. System plików	568
A.11. Elementy obsługi Gtk	570
A.12. HTML	571
A.13. HTTP	585
A.14. Pliki graficzne	588
A.15. Obsługa ustawień międzynarodowych	592
A.16. Rejestrowanie	594
A.17. Poczta	595

A.18. Operacje matematyczne	598
A.19. Obsługa sieci	602
A.20. Liczby	616
A.21. Płatności	617
A.22. PEAR	619
A.23. PHP	621
A.24. Przetwarzanie danych	629
A.25. Obliczenia naukowe	629
A.26. Strumienie	630
A.27. Struktury	632
A.28. System	634
A.29. Przetwarzanie tekstu	635
A.30. Narzędzia	637
A.31. Serwisy internetowe	640
A.32. XML	642
Dodatek B Format pakietu phpDocumentor	653
B.1. Wprowadzenie	653
B.2. Tworzenie dokumentacji za pomocą komentarzy	654
B.3. Znaczniki	655
B.4. Tabela znaczników	670
B.5. Korzystanie z narzędzia phpDocumentor	670
Dodatek C Zend Studio — szybki start	679
C.1. Wersja 3.5.x	679
C.2. O podręczniku Zend Studio Client — szybki start	679
C.3. O firmie Zend	680
C.4. Zend Studio Client: przegląd	680
C.5. Edycja plików	683
C.6. Projekty	684
C.7. Korzystanie z debugera	685
C.8. Konfiguracja pakietu Zend Studio Server w celu debugowania i tworzenia profili	688
C.9. Korzystanie z programu Profiler	688
C.10. Obsługa produktu	690
C.11. Najważniejsze własności	691
Skorowidz	693

Rozdział 3.

Język obiektowy

„Wzniosłe myśli wymagają wzniosłego języka.” — Arystofanes

3.1. Wprowadzenie

Obsługa programowania obiektowego została wprowadzona w PHP3. Choć nadawała się do pracy, była niezwykle uproszczona. Niewiele zmieniło się pod tym względem wraz z wydaniem wersji PHP4, gdyż w nowej wersji główny nacisk położono na zgodność z poprzednimi. To ogólne zapotrzebowanie na ulepszoną obsługę obiektową spowodowało, że w PHP5 ponownie zaprojektowano cały model obiektowy, dodając znaczną liczbę mechanizmów, a także zmieniając zachowanie podstawowego „obektu”.

Początkujący użytkownicy PHP dzięki lekturze tego rozdziału będą mogli zapoznać się z modelem obiektowym. Jednak zawarte tu informacje będą także użyteczne dla osób zaznajomionych już z PHP4, ponieważ wraz z wprowadzeniem wersji PHP5 uległo zmianie prawie wszystko, co ma związek z programowaniem obiektowym.

Dzięki lekturze tego rozdziału czytelnik będzie mógł poznać:

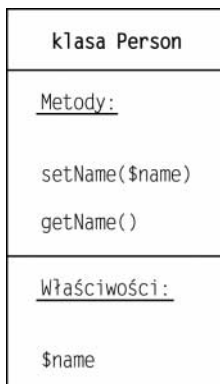
- ◆ podstawy modelu obiektowego,
- ◆ sposób tworzenia i cykl życia obiektu oraz metody jego sterowania,
- ◆ trzy główne słowa kluczowe ograniczania dostępu (`public`, `protected` i `private`),
- ◆ korzyści płynące z używania dziedziczności klas,
- ◆ wskazówki prowadzące do udanej obsługi wyjątków.

3.2. Obiekty

Programowanie obiektowe różni się od funkcjonalnego głównie tym, że dane i kod występują w nim jako jedna całość, znana pod nazwą **obiektu**. Aplikacje obiektowe zazwyczaj są podzielone na pewną liczbę obiektów, które ze sobą współdziałają. Każdy obiekt zwykle reprezentuje jeden wyodrębniony, niezależny od innych problem, i ma pewną liczbę właściwości i metod. Właściwości są **danymi** obiektu, co zasadniczo oznacza zmienne, które należą do obiektu. **Metody** — dla osób przywykłych do pracy w środowiskach funkcjonalnych — są funkcjami, które obsługuje obiekt, zaś funkcjonalność, która według przeznaczenia ma być dostępna dla innych obiektów i wykorzystywana do interakcji z nimi, jest nazywana **interfejsem** obiektu.

Rysunek 3.1 przedstawia klasę. **Klasa** jest czymś w rodzaju szablonu dla obiektu i opisuje metody i właściwości, które będzie miał obiekt danego typu. W tym przykładzie klasa reprezentuje osobę. Dla każdej osoby w aplikacji można stworzyć oddzielny egzemplarz tej klasy, reprezentujący informacje o tej osobie. Jeśli na przykład dwie osoby w naszej aplikacji noszą imiona Jan i Julia, można utworzyć dwa oddzielne egzemplarze tej klasy i dla obu wywołać metodę `setName()`, która pozwoli zainicjować zmienną `$name`, zawierającą imię danej osoby. Metody i składowe, których mogą używać inne oddziałujące obiekty, tworzą kontrakt klasy. W tym przykładzie kontraktami osoby ze światem zewnętrznym są dwie metody `set` i `get`, `setName()` i `getName()`.

Rysunek 3.1.
Schemat klasy
Person



Przedstawiony tu przykładowy kod PHP definiuje klasę, tworzy jej dwa egzemplarze, definiuje nazwę każdego egzemplarza, po czym wyświetla nazwy:

```
class Person {
    private $name;

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

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

```
};

$julia = new Person();
$julia->setName("Julia");

$jan = new Person();
$jan->setName("Jan");

print $julia->getName() . "\n";
print $jan->getName() . "\n";
```

3.3. Deklarowanie klasy

Wnioskując z prezentowanego przed chwilą przykładu można stwierdzić, że zdefiniowanie klasy (szablону obiektu) nie jest niczym trudnym. Wystarczy użyć słowa kluczowego `class`, nadać klasie nazwę, po czym wymienić wszystkie metody i właściwości, które powinien mieć egzemplarz tej klasy:

```
class MyClass {
    ... // Lista metod
    ...
    ... // Lista właściwości
    ...
}
```

Jak widać, na początku definicji właściwości `$name` użyte zostało słowo kluczowe `private`. Zostanie ono omówione szczegółowo w dalszej części książki; w tym przypadku oznacza ono, że do `$name` mogą mieć dostęp tylko metody z tej klasy. Zmusza to każdego, kto chciałby pobrać lub ustawić tę właściwość, do posłużenia się metodami `getName()` i `setName()`, które reprezentują interfejs klasy przeznaczony do użytku przez inne obiekty lub kod źródłowy.

3.4. Słowo kluczowe `new` i konstruktory

Egzemplarze klas są tworzone przy użyciu słowa kluczowego `new`. W poprzednim przykładzie nowy egzemplarz klasy `Person` został utworzony przy pomocy instrukcji `$julia = new Person();`. Podczas wywołania `new` zostaje przydzielony nowy obiekt, z własnymi kopiami właściwości zdefiniowanymi we wskazanej klasie, po czym przywołany zostaje konstruktor obiektu, jeśli tylko został zdefiniowany. Konstruktor to metoda o nazwie `__construct()`, która jest automatycznie przywoływana przez słowo kluczowe `new` po utworzeniu obiektu. Zazwyczaj służy do automatycznego wykonywania różnych inicjalizacji, takich jak inicjalizacja właściwości. Konstruktory mogą również pobierać argumenty; w takim przypadku w momencie napisania instrukcji `new` trzeba również przekazać konstruktorowi parametry funkcji, umieszczając je między nawiasami.

W PHP4, podobnie jak w C++, zamiast używać metody `__construct()` jako nazwy konstruktora, definiowało się metodę, nadając jej taką samą nazwę jak ta, którą posiadała klasa. W PHP5 możliwość ta wciąż istnieje, jednak dla nowych aplikacji należy używać nowej, zunifikowanej konwencji nazywania konstruktorów.

Nasz poprzedni przykład można napisać w inny sposób, tak by w wierszu `new` kod przekazywał imiona osób:

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

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

    private $name;
};

$julia = new Person("Julia") . "\n";
$jan = new Person("Jan") . "\n";

print $julia->getName();
print $jan->getName();
```

Wynik zwrócony przez ten kod będzie identyczny z tym z poprzedniego przykładu.



Ponieważ konstruktor nie może zwracać wartości, najczęstszą praktyką zgłaszania błędu ze środka konstruktora jest rzucenie wyjątku.

3.5. Destruktory

Destruktory to odwrotność konstruktorów. Wywoływane są one w momencie usuwania obiektu (na przykład gdy do danego obiektu nie ma już referencji). Ponieważ PHP pilnuje, by po każdym żądaniu wszystkie zasoby zostały zwolnione, destruktory mają ograniczone znaczenie. Mimo to wciąż mogą być przydatne do wykonywania określonych czynności, takich jak opróżnianie zasobu lub informacji o rejestracji w momencie usuwania obiektu. Destruktor może być wywołany w dwóch sytuacjach: podczas wykonywania skryptu, kiedy niszczone są wszystkie odwołania do obiektu, lub w momencie osiągnięcia końca skryptu i zakończenia wykonywania przez PHP żądania. Druga sytuacja jest dość niedogodna, ponieważ jest się uzależnionym od pewnych obiektów, których destruktory mogły już zostać wywołane, wskutek czego obiekty te nie są już dostępne. Stąd też destruktory należy używać ostrożnie i nie uzależniać ich od innych obiektów.

Definiowanie destruktora polega na dodaniu do klasy metody `__destruct()`:

```
class MyClass {
    function __destruct()
    {
        print "Niszczony jest obiekt typu MyClass\n";
    }
}

$objj = new MyClass();
$objj = NULL;
```

W rezultacie skrypt wyświetli:

```
Niszczony jest obiekt typu MyClass
```

W tym przykładzie dojdzie do instrukcji `$objj = NULL;` powoduje usunięcie jedynego uchwytu do obiektu, w następstwie czego zostaje wywołany destruktor, a sam obiekt ulega zniszczeniu. Destruktor zostałby wywołany także bez ostatniego wiersza kodu, tym razem jednak podczas kończenia wykonywania żądania.



PHP nie gwarantuje dokładnego czasu wywołania destruktoru. Może to być kilka instrukcji po wykonaniu ostatniego odwołania do obiektu. Z tego właśnie względu należy uważać, by nie pisać aplikacji w sposób, który mógłby doprowadzić do kłopotów.

3.6. Odwoływanie się do metod i właściwości przy użyciu zmiennej `$this`

Podczas wykonywania metody obiektu definiowana jest automatycznie specjalna zmienna o nazwie `$this`, która jest odwołaniem do samego obiektu. Używając tej zmiennej i notacji `->` można odwoływać się do metod i właściwości tego obiektu. Na przykład można uzyskać dostęp do właściwości `$name` przy użyciu instrukcji `$this->name` (należy zauważyć, że przed nazwą właściwości nie używa się znaku `$`). Metody obiektu można udostępnić w ten sam sposób: na przykład z wnętrza jednej z metod klasy osoby (`Person`) można wywołać funkcję `getName()`, wpisując instrukcję `$this->getName()`.

3.6.1. Właściwości `public`, `protected` i `private`

W programowaniu obiektowym kluczowym paradygmatem jest hermetyzacja i kontrola dostępu do właściwości obiektu (nazywanych także zmiennymi składowymi). W najpopularniejszych językach obiektowych można spotkać trzy główne słowa kluczowe kontroli dostępu: `public`, `protected` i `private`.

Definiując składową klasy w definicji klasy, programista musi przed zadeklarowaniem samej składowej umieścić jedno z tych trzech słów. Osoby zaznajomione z modelem obiektowym PHP3 lub 4 wiedzą, że w tamtych wersjach wszystkie składowe klasy były definiowane ze słowem kluczowym `var`. W PHP5 jego odpowiednikiem jest słowo kluczowe `public`, choć etykieta `var` została zachowana w celu zapewnienia kompatybilności ze starszymi wersjami programu. Składnia ta jest jednak przestarzała, dlatego namawiamy do przekształcenia swych skryptów i dostosowania ich do nowych wymogów:

```
class MyClass {
    public $publicMember = "Składowa publiczna";
    protected $protectedMember = "Składowa chroniona";
    private $privateMember = "Składowa prywatna";

    function myMethod(){
        //...
    }
}

$obj = new MyClass();
```

Ten przykład będzie jeszcze rozbudowywany, w celu zademonstrowania użycia tych modyfikatorów dostępu.

Najpierw jednak podajmy definicję każdego z modyfikatorów dostępu:

- ◆ **public**. Dostęp do składowych publicznych można uzyskać spoza obiektu przy użyciu kodu `$obj->publicMember` oraz z wnętrza metody `myMethod` przy użyciu specjalnej zmiennej `$this` (na przykład `$this->publicMember`). Jeśli składową publiczną odziedziczy inna klasa, zastosowanie mają te same zasady, co oznacza, że dostęp do niej można uzyskać tak z obiektów spoza klasy pochodnej, jak i z wnętrza metod do niej należących.
- ◆ **protected**. Dostęp do składowych chronionych można uzyskać tylko z wnętrza metody obiektu — na przykład `$this->protectedMember`. Jeśli inna klasa odziedziczy składową chronioną, zastosowanie mają te same zasady, można zatem do niej sięgnąć z wnętrza metody pochodnego obiektu poprzez specjalną zmienną `$this`.
- ◆ **private**. Składowe prywatne są podobne do składowych chronionych, ponieważ dostęp do nich jest możliwy tylko z wnętrza metody obiektu. Nie są natomiast dostępne z metod obiektów pochodnych. Ponieważ właściwości prywatne nie są widoczne z klas pochodnych, dwie pokrewne ze sobą klasy mogą deklarować te same właściwości prywatne. Każda klasa będzie „widzieć” tylko własną prywatną kopię, która nie jest powiązana z innymi.

Zazwyczaj w przypadku składowych, które mają być dostępne spoza obiektu, należy używać słowa `public`, zaś dla tych, które są — według logiki obiektu — wewnętrzne, słowa `private`. Z kolei słowa `protected` należy używać do tych składowych, które według logiki obiektu są wewnętrzne, ale tylko wtedy, gdy rozsądne jest ich zignorowanie w klasach pochodnych:


```

class MyDbConnectionClass {
    public $queryResult;
    protected $dbHostname = "localhost";
    private $connectionHandle;

    // ...
}

class MyFooDotComDbConnectionClass extends MyDbConnectionClass {
    protected $dbHostname = "foo.com";
}

```

Ten przykład niepełnego kodu przedstawia typowe użycie każdego z trzech modyfikatorów dostępu. Klasa z naszego przykładu zarządza połączeniem z bazą danych, włączając w to kierowane do bazy zapytania:

- ♦ Uchwyt połączenia z bazą danych znajduje się w składowej `private`, ponieważ wykorzystywany jest tylko przez wewnętrzną logikę klasy i nie powinien być dostępny dla użytkownika tej klasy.
- ♦ W tym przykładzie nazwa hosta bazy danych nie zostaje ujawniona użytkownikowi klasy `MyDbConnectionClass`. Aby to anulować, programista może stworzyć klasę pochodną i zmienić wartość.
- ♦ Sam wynik zapytania powinien być dostępny dla programisty i dlatego został zadeklarowany jako publiczny.

Należy zauważyć, że modyfikatory dostępu zostały zaprojektowane w sposób, który pozwala na to, by klasy (lub bardziej dokładnie ich interfejs pozwalający na komunikację ze „światem zewnętrznym”) zawsze zachowywały podczas dziedziczenia relację „jest”. W takim przypadku, jeśli klasa macierzysta zadeklaruje składową jako publiczną, dziedziczący potomek musi również zadeklarować ją jako publiczną. W przeciwnym razie nie byłby w stosunku do swojej klasy nadrzędnej w relacji „jest”, która oznacza, że cokolwiek zrobi się z klasą macierzystą, można to także zrobić z potomną.

3.6.2. Metody `public`, `protected` i `private`

Modyfikatorów dostępu można również używać w połączeniu z metodami obiektu; zasady są w tym przypadku takie same jak poprzednio:

- ♦ Metody publiczne (`public`) można wywoływać z każdego zasięgu.
- ♦ Metody chronione (`protected`) można wywoływać tylko z wnętrza jednej z metod klasy lub klasy dziedziczącej.
- ♦ Metody prywatne (`private`) można również wywoływać z wnętrza jednej z metod klasy obiektu, ale nie z klasy dziedziczącej. Podobnie jak w przypadku właściwości, metody `private` mogą być ponownie definiowane przez klasy dziedziczące. Każda klasa będzie „widzieć” własną wersję metody:

```

class MyDbConnectionClass {
    public function connect()
    {
        $conn = $this->createDbConnection();
    }
}

```

```

        $this->setDbConnection($conn);
        return $conn;
    }

    protected function createDbConnection()
    {
        return mysql_connect("localhost");
    }

    private function setDbConnection($conn)
    {
        $this->dbConnection = $conn;
    }

    private $dbConnection;
}

class MyFooDotComDbConnectionClass extends MyDbConnectionClass {
    protected function createDbConnection()
    {
        return mysql_connect{"foo.com"};
    }
}

```

Ten przykładowy zarys kodu może być użyty w stosunku do klas połączeń z bazą danych. Metoda `connect()` ma być wywoływana przez kod zewnętrzny. Metoda `createDbConnection()` jest metodą wewnętrzną, ale pozwala na dziedziczenie z klasy oraz na jej zmianę, dlatego jest oznaczona słowem `protected`. Metoda `setDbConnection()` jest całkowicie wewnętrzną względem klasy i dlatego także ona oznaczona jest jako `private`.



W przypadku braku modyfikatora dostępu dla metody domyślnie używana jest opcja `public`. Z tego powodu w pozostałych rozdziałach słowo kluczowe `public` będzie często pomijane.

3.6.3. Właściwości statyczne

Wiadomo już, że w klasach można deklarować właściwości. Każdy egzemplarz klasy (tj. obiekt) ma własną kopię tych właściwości. Jednak klasa może również zawierać **właściwości statyczne**. W przeciwieństwie do zwykłych właściwości, te należą do samej klasy, a nie do któregośkolwiek z jej egzemplarzy. Stąd są one często nazywane **właściwościami klasy**, w przeciwieństwie do właściwości obiektu lub egzemplarza. Można o nich myśleć również jako o zmiennych globalnych, które znajdują się w klasie, ale poprzez tę klasę są dostępne wszędzie.

Właściwości statyczne definiuje się przy użyciu słowa kluczowego `static`:

```

class MyClass {
    static $myStaticVariable;
    static $myInitializedStaticVariable = 0;
}

```

Aby uzyskać dostęp do właściwości statycznych, należy podać nazwę właściwości z klasą, w której się ona znajduje:

```
MyClass::$myInitializedStaticVariable++;  
print MyClass::$myInitializedStaticVariable;
```

Wynikiem wykonania kodu z tego przykładu będzie wyświetlenie liczby 1.

W przypadku pobierania składowej z wnętrza jednej z metod klasy, do właściwości można również odwoływać się poprzedzając ją specjalną nazwą klasy `self`. Jest to skrót oznaczający klasę, do której należy metoda:

```
class MyClass {  
    static $myInitializedStaticVariable = 0;  
  
    function myMethod()  
    {  
        print self::$myInitializedStaticVariable;  
    }  
}  
  
$obj = new MyClass();  
$obj->myMethod;
```

Kod z tego przykładu wyświetli liczbę 0.

Niektórzy z czytelników zadają sobie prawdopodobnie pytanie, czy właściwości statyczne są naprawdę przydatne.

Jednym z przykładów ich użycia jest przypisanie niepowtarzalnego numeru identyfikacyjnego do wszystkich egzemplarzy klasy:

```
class MyUniqueIDClass {  
    static $idCounter = 0;  
  
    public $uniqueId;  
  
    function __construct()  
    {  
        self::$idCounter++;  
        $this->uniqueId = self::$idCounter;  
    }  
}  
  
$obj1 = new MyUniqueIdClass();  
print $obj1->uniqueId . "\n";  
$obj2 = new MyUniqueIdClass();  
print $obj2->uniqueId . "\n";
```

Wynikiem tego skryptu będzie:

```
1  
2
```

Wartość właściwości `$uniqueId` pierwszego obiektu równa się 1, zaś drugiego — 2.

Jeszcze lepszym przykładem wykorzystania właściwości statycznej jest wzorzec Singleton, przedstawiony w następnym rozdziale.

3.6.4. Metody statyczne

Podobnie jak w przypadku właściwości, w PHP można definiować jako **statyczne** także metody. Taka metoda jest częścią klasy i nie jest przypisana do żadnego określonego egzemplarza obiektu i jego właściwości. Stąd nie jest w niej dostępna zmienna `$this`, zaś klasa jest dostępna przy użyciu słowa `self`. Ponieważ metody statyczne nie są przypisane do żadnego określonego obiektu, można je wywoływać bez tworzenia egzemplarza obiektu przy użyciu składni `nazwa_klasy::metoda()`. Można również wywoływać je z egzemplarza obiektu przy użyciu `$this->metoda()`, jednak `$this` nie zostanie zdefiniowane w wywoływanej metodzie. Dla większej przejrzystości zamiast składni `$this->metoda()` winno się używać `self::metoda()`.

Oto przykład:

```
class PrettyPrinter {
    static function printHelloWorld()
    {
        print "Witaj świecie";
        self::printNewline();
    }

    static function printNewline()
    {
        print "\n";
    }
}

PrettyPrinter::printHelloWorld();
```

Przykład wyświetla łańcuch "Witaj świecie", po którym następuje znak nowego wiersza.

Choć przykład ten jest raczej mało przydatny, pozwala zaobserwować, że funkcję `printHelloWorld()` można wywoływać w klasie bez tworzenia egzemplarza obiektu przy użyciu nazwy klasy, zaś sama metoda statyczna może wywołać następną metodę statyczną `printNewline()` przy użyciu notacji `self::`. Istnieje możliwość wywołania metody statycznej klasy nadrzędnej przy użyciu notacji `parent::`, co zostanie omówione nieco dalej w tym samym rozdziale.

3.7. Stałe klasy

Stałe globalne istniały w PHP od dawna. Można je było definiować przy użyciu funkcji `define()`, która została opisana w rozdziale 2., zatytułowanym „Podstawy języka PHP5”. Dzięki ulepszonej obsłudze hermetyzacji w PHP5 można zdefiniować stałe wewnątrz klas. Podobnie jak w przypadku składowych statycznych, należą one do klasy, a nie do egzemplarza klasy. W stałych klasy zawsze rozróżniania jest wielkość liter. Składnia deklaracji jest intuicyjna, a sposób dostępu do stałych przypomina dostęp do składowych statycznych:

```
class MyColorEnumClass {
    const RED = "Czerwony";
    const GREEN = "Zielony";
    const BLUE = "Niebieski";

    function printBlue()
    {
        print self::BLUE;
    }
}

print MyColorEnumClass::RED;
$objj = new MyColorEnumClass();
$objj->printBlue();
```

Ten kod wyświetli łańcuch "Czerwony", a po nim "Niebieski". Przykład demonstruje możliwość udostępniania stałej z wnętrza metody klasy przy użyciu słowa kluczowego `self` i poprzez nazwę klasy "MyColorEnumClass".

Jak można wnioskować z nazwy, stałe mają stały, niezmienny charakter i po zdefiniowaniu nie można ich zmieniać ani usuwać. Do popularnych zastosowań stałych należy definiowanie wyliczeń, co zostało zaprezentowane w ostatnim przykładzie, lub pewnej wartości konfiguracyjnej, takiej jak nazwa użytkownika bazy danych, której aplikacja nie powinna móc zmienić.



Podobnie jak w przypadku zmiennych globalnych, przyjęte jest pisanie nazw stałych wielkimi literami.

3.8. Klonowanie obiektów

W przypadku tworzenia obiektu przy użyciu słowa kluczowego `new`, zwrócona wartość jest uchwytem do obiektu lub — mówiąc inaczej — **numerem identyfikacyjnym** obiektu. To odróżnia PHP5 od jego wcześniejszej wersji, w której wartość była po prostu obiektem. Nie oznacza to, że składnia wywołania metod lub udostępniania właściwości uległa zmianie, ale zmieniła się semantyka kopiowania obiektów.

Rozważmy następujący kod:

```
class MyClass {
    public $var = 1;
}

$objj1 = new MyClass();
$objj2 = $objj1;
$objj2->var = 2;
print $objj1->var;
```

W PHP4 ten kod zwróciłby liczbę 1, ponieważ do zmiennej `$objj2` jest przypisana wartość obiektu `$objj1`, tworząc w ten sposób kopię i pozostawiając `$objj1` bez zmian.

Jednak w PHP5 `$obj1` jest uchwytym obiektu (jest numerem identyfikacyjnym), więc do `$obj2` jest kopiowany uchwyt. Podczas zmiany zmiennej `$obj2`, można w rzeczywistości zmienić sam obiekt, do którego zmienna `$obj1` się odnosi. Wykonanie tego fragmentu kodu powoduje otrzymanie liczby 2.

Czasami jednak naprawdę chodzi o utworzenie kopii obiektu. Jak można to osiągnąć? Rozwiązaniem jest konstrukcja językowa `clone`. Ten wbudowany operator tworzy automatycznie nowy egzemplarz obiektu z jego własną kopią właściwości. Wartości właściwości zostają skopiowane w swej pierwotnej postaci. Dodatkowo można zdefiniować metodę `__clone()`, która jest wywoływana na nowo utworzonych obiektach w celu wykonania wszystkich końcowych operacji.



Referencje po skopiowaniu pozostają referencjami i nie wykonują pełnej kopii zawartości struktury. Oznacza to, że jeśli jedna z właściwości wskazuje przez referencję na inną zmienną (po tym jak została ona przypisana przez referencję), to po automatycznym klonowaniu, obiekt sklonowany będzie wskazywał na tę samą zmienną.

Zmiana w ostatnim przykładzie wiersza `$obj2 = $obj1;` na `$obj2 = clone $obj1;` spowodowałoby przypisanie do `$obj2` uchwytu do nowej kopii `$obj1`, wskutek czego otrzymanym wynikiem byłaby liczba 1.

Jak zostało już wcześniej wspomniane, metodę `__clone()` można implementować dla wszystkich klas. Jest ona wywołana po utworzeniu nowego (skopiowanego) obiektu, a sklonowany obiekt jest dostępny przy użyciu zmiennej `$this`.

Poniżej pokazano przykład typowej sytuacji, w której warto zaimplementować metodę `__clone()`. Powiedzmy, że mamy obiekt zawierający uchwyt do pliku. Nie chcemy, by nowy obiekt wskazywał na ten sam uchwyt do pliku. Powinien otwierać nowy plik, aby mieć własną, prywatną kopię:

```
class MyFile {
    function setFileName($file_name)
    {
        $this->file_name = $file_name;
    }

    function openFileForReading()
    {
        $this->file_handle = fopen($this->file_name, "r");
    }

    function __clone()
    {
        if ($this->file_handle) {
            $this->file_handle = fopen($this->file_name, "r");
        }
    }

    private $file_name;
    private $file_handle = NULL;
}
```

Choć jest to zaledwie fragment kodu, pozwala zaobserwować sposób kontroli nad procesem klonowania. Zmienna `$file_name` zostaje tutaj skopiowana z oryginalnego obiektu, ale jeśli ten ma uchwyt otwartego pliku, który został skopiowany do sklonowanego obiektu, nowa kopia obiektu utworzy własną kopię uchwytu pliku, tworząc nowe połączenie z tym plikiem.

3.9. Polimorfizm

Polimorfizm jest prawdopodobnie najważniejszym zagadnieniem związanym z programowaniem obiektowym. Użycie klas i dziedziczności ułatwia opisanie rzeczywistych sytuacji, czego nie da się powiedzieć o zwykłym zbiorze funkcji i danych. Dziedziczność ułatwia także rozwój projektów przez ponowne wykorzystanie kodu. Ponadto, pisząc duży i rozszerzalny kod, zazwyczaj zależy nam na jak najmniejszej liczbie instrukcji sterowania przepływem (takich jak instrukcje `if()`). Odpowiedzią na te wszystkie potrzeby — i na wiele innych — jest polimorfizm.

Rozważmy następujący kod:

```
class Cat {
    function miau()
    {
        print "miau";
    }
}

class Dog {
    function wuff()
    {
        print "hau";
    }
}

function printTheRightSound($obj)
{
    if ($obj instanceof Cat) {
        $obj->miau();
    } else if {$obj instanceof Dog} {
        $obj->wuff();
    } else {
        print "Błąd: Przekazano zły rodzaj obiektu";
    }
    print "\n";
}

printTheRightSound(new Cat());
printTheRightSound(new Dog());
```

W wyniku otrzymamy:

```
miau
hau
```

Nietrudno zauważyć, że ten przykład nie jest rozszerzalny. Załóżmy, że chcemy go rozszerzyć o odgłosy trzech kolejnych zwierząt. Oznaczałoby to dodanie do funkcji `printTheRightSound()` kolejnego bloku `else if`, aby sprawdzać, czy dany obiekt jest egzemplarzem jednego z tych nowych gatunków zwierząt. Następnie należałoby dodać kod, który wywoływałby metodę każdego odgłosu.

Problem ten rozwiązuje zastosowanie polimorfizmu z użyciem dziedziczności, gdyż pozwala na dziedziczenie z klasy nadrzędnej, co oznacza dziedziczenie wszystkich jej metod i właściwości oraz utworzenie w ten sposób związku „jest”.

Aby dostosować poprzedni przykład do wspomnianych potrzeb, utworzona zostanie w nim nowa klasa, o nazwie `Animal`, z której będą dziedziczyć wszystkie zwierzęta, tworząc relację „jest” od określonych rodzajów zwierząt, na przykład psa (klasa `Dog`) do klasy nadrzędnej (lub przodka) `Animal`.

Dziedziczenie wykonuje się przy użyciu słowa kluczowego `extends`:

```
class Potomek extends Przodek {  
    ...  
}
```

Oto jak można przepisać poprzedni przykład z wykorzystaniem dziedziczności:

```
class Animal {  
    function makeSound()  
    {  
        print "Błąd: Ta metoda powinna być ponownie zaimplementowana w klasach  
        ➔ potomnych";  
    }  
}  
  
class Cat extends Animal {  
    function makeSound()  
    {  
        print "miau";  
    }  
}  
  
class Dog extends Animal {  
    function makeSound()  
    {  
        print "hau";  
    }  
}  
  
function printTheRightSound($obj)  
{  
    if ($obj instanceof Animal) {  
        $obj->makeSound();  
    } else {  
        print "Błąd: Przekazano zły rodzaj obiektu";  
    }  
    print "\n";  
}
```



```
printTheRightSound(new Cat());  
printTheRightSound(new Dog());
```

Wynikiem będą łańcuchy:

```
miau  
hau
```

Warto zauważyć, że tym razem, bez względu na liczbę gatunków zwierząt dodanych do przykładu, nie trzeba będzie wprowadzać żadnych zmian w funkcji `printTheRightSound()`. Wszystkimi zmianami zajmie się warunek `instanceof Animal` oraz wywołanie `$obj->makeSound()`.

Ten przykład można jeszcze poprawiać. W PHP dostępne są pewne modyfikatory, które mogą zwiększyć kontrolę nad procesem dziedziczenia. Zostały one omówione szczegółowo w dalszej części tego rozdziału. Na przykład klasę `Animal` i jej metodę `makeSound()` można oznaczyć jako abstrakcyjną przy użyciu modyfikatora `abstract`. Pozwoli to uniknąć dodawania nic nieznaczących implementacji definicji metody `makeSound()` w klasie `Animal`, ale wymusi również jej implementację we wszystkich klasach dziedziczących. Dla metody `makeSound()` można ponadto podać modyfikatory dostępu. Na przykład modyfikator `public` będzie oznaczać, że metodę tę można będzie wywoływać z każdego miejsca w kodzie.



PHP nie obsługuje dziedziczenia wielokrotnego, które występuje w języku C++. Dostarcza za to inne rozwiązanie prowadzące do utworzenia więcej niż jednej relacji „jest” dla danej klasy, wykorzystujące interfejsy podobne do tych z języka Java; zostaną one omówione w dalszej części tego rozdziału.

3.10. parent:: i self::

W PHP obsługiwane są dwie zarezerwowane nazwy klas, które ułatwiają pisanie aplikacji obiektowych. Nazwa `self::` odnosi się do bieżącej klasy i zazwyczaj służy do pobierania składników statycznych, metod i stałych. Nazwa `parent::` odnosi się do klasy nadrzędnej i najczęściej jest wykorzystywana, kiedy zachodzi potrzeba wywołania konstruktora lub metod nadrzędnych. Może również służyć do pobierania składowych i stałych. Nazwy `parent::` należy używać zamiast nazwy klasy nadrzędnej, ponieważ ułatwia to zmianę hierarchii klasy. Dzięki temu nazwa przodka nie jest wpisana na stałe w kod.

Poniższy przykład wykorzystuje nazwy `parent::` i `self::` w celu udostępnienia klas `Child` i `Ancestor`:

```
class Ancestor {  
    const NAME = "Przodek";  
    function __construct()  
    {  
        print "W konstruktorze " . self::NAME . "\n";  
    }  
}
```

```
class Child extends Ancestor {
    const NAME = "Potomek";
    function __construct()
    {
        parent::__construct();
        print "W konstruktorze " . self::NAME . "\n";
    }
}

$objj = new Child();
```

W wyniku wykonania tego przykładowego kodu na ekranie pojawi się:

```
W konstruktorze Przodek
W konstruktorze Potomek
```

Tych dwóch nazw klas warto używać wszędzie tam, gdzie jest to tylko możliwe.

3.11. Operator instanceof

Operator `instanceof` został dodany do PHP jako pewien rodzaj syntaktycznego ozdobnika, który miał zastąpić wbudowaną i aktualnie przestarzałą funkcję `is_a()`. W przeciwieństwie do niej, sposób użycia `instanceof` przypomina zastosowanie logicznego operatora binarnego:

```
class Rectangle {
    public $name = __CLASS__;
}

class Square extends Rectangle {
    public $name = __CLASS__;
}

class Circle {
    public $name = __CLASS__;
}

function checkIfRectangle($shape)
{
    if ($shape instanceof Rectangle) {
        print $shape->name;
        print " jest prostokątem\n";
    }
}

checkIfRectangle(new Square());
checkIfRectangle(new Circle());
```

Ten mały program wyświetli komunikat `'Square jest prostokątem\n'`. Warto zwrócić uwagę na użycie stałej `__CLASS__`. Jest to specjalna stała, która zwraca nazwę bieżącej klasy.

Jak zostało już wcześniej wspomniane, `instanceof` jest operatorem i dlatego może być używany w wyrażeniach w połączeniu z innymi operatorami (na przykład z operatorem negacji `!`). Umożliwia to proste napisanie funkcji `checkIfNotRectangle`:

```
function checkIfNotRectangle($shape)
{
    if (!$shape instanceof Rectangle) {
        print $shape->name;
        print " nie jest prostokątem\n";
    }
}
```

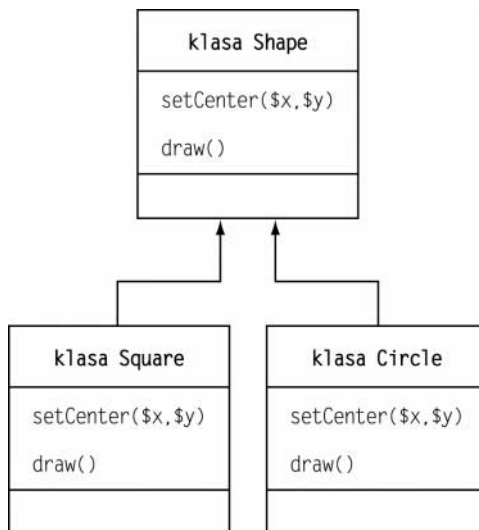


Operator `instanceof` sprawdza również, czy obiekt implementuje interfejs (który także jest klasyczną relacją „jest”). Interfejsy zostały omówione w dalszej części rozdziału.

3.12. Metody i klasy abstrakcyjne

Podczas projektowania hierarchii klas warto pozostawić niektóre metody do implementacji przez klasy dziedziczące. Załóżmy na przykład, że mamy klasę o hierarchii przedstawionej na rysunku 3.2.

Rysunek 3.2.
Hierarchia klasy



Jedną z rozsądnych możliwości jest zaimplementowanie metody definiującej środek figury `setCenter($x, $y)` w klasie `Shape` i pozostawienie implementacji rysującej figury metody `draw()` do konkretnych klas `Square` i `Circle`. W tym celu metodę `draw()` należy zdefiniować jako metodę `abstract`. Dzięki temu poinformujemy PHP, że specjalnie nie implementujemy jej w klasie `Shape`. Klasa `Shape` stanie się klasą abstrakcyjną, co oznacza, że nie jest to klasa o pełnej funkcjonalności i można z niej tylko dziedziczyć. Klasa abstrakcyjna nie może mieć egzemplarzy. Modyfikator `abstract`

można zdefiniować dla dowolnej liczby metod w klasie, jednak wystarczy, by choć jedna metoda z klasy została zadeklarowana jako abstrakcyjna, by jako taką należało zadeklarować całą klasę. Ta podwójna definicja pozwala zdefiniować klasę jako `abstract`, nawet jeśli nie zawiera ona żadnych abstrakcyjnych metod, i wymusza, by klasa posiadająca metody abstrakcyjne sama została zadeklarowana jako abstrakcyjna, co pozwoli na lepsze zrozumienie intencji programisty.

Poprzedni diagram klas można by przetłumaczyć na następujący kod PHP:

```
abstract class Shape {
    function setCenter($x, $y) {
        $this->x = $x;
        $this->y = $y;
    }

    abstract function draw();
    protected $x, $y;
}

class Square extends Shape {
    function draw()
    {
        // Tutaj kod rysujący kwadrat
        ...
    }
}

class Circle extends Shape {
    function draw()
    {
        // Tutaj kod rysujący koło
        ...
    }
}
```

Jak widać, metoda abstrakcyjna `draw()` nie zawiera żadnego kodu.



W przeciwieństwie do niektórych języków, nie można definiować metody `abstract` z domyślną implementacją. W PHP metoda może być abstrakcyjna, czyli bez kodu, lub w pełni zdefiniowana.

3.13. Interfejsy

Dziedziczenie klas umożliwia opisywanie relacji rodzic-potomek między klasami. Można na przykład mieć klasę bazową kształtów `Shape`, z której wywodzą się klasy kwadratów `Square` i kół `Circle`. Często zachodzi jednak konieczność dodania do klas dodatkowych „interfejsów”, co zasadniczo oznacza dodatkowe warunki, do których musi zastosować się klasa. W języku C++ osiąga się to przy użyciu dziedziczenia wielokrotnego i wprowadzania z dwóch klas. W PHP alternatywą dla dziedziczenia

wielokrotnego są interfejsy, umożliwiające określanie dodatkowych warunków, jakie musi spełniać klasa. Interfejs definiuje się podobnie do klasy, ale zawiera on tylko prototypy funkcji (bez implementacji) oraz stałe. Każda klasa, która implementuje ten interfejs automatycznie, będzie miała zdefiniowane stałe interfejsu i, jako klasa realizująca, musi dostarczyć definicje funkcji dla prototypów funkcji interfejsu, z których wszystkie są metodami `abstract` (chyba że jako `abstract` została zadeklarowana klasa realizująca).

Aby dokonać implementacji interfejsu, należy zastosować następującą składnię:

```
class A implements B, C, ... {  
    ...  
}
```

Klasy realizujące dany interfejs pozostają w relacji „jest” do interfejsu (`instanceof`); jeśli na przykład klasa `A` implementuje interfejs `myInterface`, poniższy kod spowoduje wydrukowanie `'$obj jest egzemplarzem myInterface'`:

```
$obj = new A();  
if ($obj instanceof myInterface) {  
    print '$obj jest egzemplarzem myInterface';  
}
```

Kod naszego kolejnego przykładu definiuje interfejs o nazwie `Loggable`. Będzie on realizowany przez klasy w celu zdefiniowania informacji, które ma rejestrować funkcja `MyLog()`. Obiekty klas, które nie zaimplementują tego interfejsu i zostaną przekazane do funkcji `MyLog()`, spowodują wyświetlenie komunikatu o błędzie:

```
interface Loggable {  
    function logString();  
}  
  
class Person implements Loggable {  
    private $name, $address, $idNumber, $age;  
    function logString() {  
        return "klasa Person: nazwisko = $this->name, ID = $this->idNumber\n";  
    }  
}  
  
class Product implements Loggable {  
    private $name, $price, $expiryDate;  
    function logString() {  
        return "klasa Product: nazwa = $this->name, cena = $this->price\n";  
    }  
}  
  
function MyLog($obj) {  
    if ($obj instanceof Loggable) {  
        print $obj->logString();  
    } else {  
        print "Błąd: Obiekt nie obsługuje interfejsu Loggable\n";  
    }  
}
```

```
$person = new Person();  
//...  
$product = new Product();  
  
MyLog($person);  
MyLog($product);
```



Interfejsy są domyślnie publiczne, dlatego w definicji interfejsu nie można określać modyfikatorów metody dla prototypów metod.



Nie wolno dopuszczać do implementacji interfejsów, które ze sobą kolidują (na przykład interfejsy definiujące te same stałe lub metody).