helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

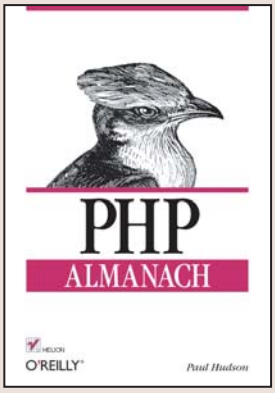
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

PHP. Almanach



Autor: Paul Hudson
Tłumaczenie: Robert Górczyński
ISBN: 83-246-0348-4
Tytuł oryginału: PHP in a Nutshell
Format: B5, stron: 384

Język PHP zyskał już liczne grono zadowolonych użytkowników. Jest wykorzystywany w witrynach internetowych do obsługi formularzy, wyświetlania treści z bazy danych i do wielu innych zadań. Co przysporzyło mu tak wielkiej popularności? Prosta składnia, wielkie możliwości i doskonała dokumentacja dostępna w sieci? Na pewno tak, ale jedną z jego największych zalet jest bezpłatna dystrybucja. Najnowsza wersja PHP, oznaczona numerem 5, to w pełni obiektowy język programowania pozwalający na korzystanie z niemal wszystkich nowoczesnych internetowych rozwiązań technologicznych – języka XML, usług sieciowych czy protokołu SOAP.

Książka „PHP. Almanach" to kompletny przewodnik po najnowszej wersji tego języka. Znajdują się w niej informacje na temat programowania obiektowego, tworzenia elementów dynamicznych witryn WWW i zabezpieczania aplikacji przed dostępem osób niepowołanych. Zamieszczono tu również opisy kilku najpopularniejszych rozszerzeń języka PHP, które mogą okazać się bardzo przydatne przy tworzeniu aplikacji internetowych.

W książce omówiona między innymi:

- Instalacja PHP w Windows i Linuksie
- Podstawowe elementy języka PHP
- Programowanie obiektowe
- Obsługa formularzy na stronach WWW
- Zarządzanie sesjami i plikami cookie
- Buforowanie danych wyjściowych
- Operacje na plikach i bazach danych
- Generowanie grafiki, plików PDF i SWF
- Obsługa plików XML
- Dystrybucja aplikacji
- Testowanie i usuwanie błędów
- Optymalizacja kodu

To obowiązkowa lektura każdego programisty PHP

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

Spis treści

Wprowadzenie **9**

1. Wprowadzenie do PHP **13**
Historia PHP 13
Zalety PHP 14
Uzyskiwanie pomocy 16
Uzyskanie certyfikatu 20
Źródła wiedzy o PHP 20

2. Instalacja PHP **23**
Instalacja w systemie Windows 23
Instalacja w systemie Unix 26
Testowanie swojej konfiguracji 29
Konfiguracja systemu 30

3. Interpreter PHP **31**
Uruchamianie skryptów PHP 31
Rozszerzanie PHP 32
PEAR 33
Nieprawidłowe przerwanie wykonywania skryptu 34

4. Język PHP **37**
Podstawy PHP 37
Zmienne 38
Wolne przestrzenie 40
Heredoc 40
Krótkie wprowadzenie do typów zmiennych 41
Bloki kodu 41
Otwarcie i zamknięcie bloków kodu 41
Komentarze 42

Instrukcje warunkowe 42
Case Switching 44
Pętle 46
Nieskończone pętle 48
Specjalne słowa kluczowe pętli 48
Pętle wewnątrz pętli 49
Przetwarzanie trybów mieszanych 51
Dołączanie innych plików 51
Funkcje 53

5. Zmienne i stałe ... 61
Typy danych 61
Wartość true lub false 62
Łańcuchy 62
Liczby całkowite 63
Liczby zmiennoprzecinkowe 64
Automatyczna konwersja typu 65
Sprawdzenie, czy zmienna jest ustalona: isset() 66
Zasięg zmiennej 67
Zmienne zmiennych 67
Tablice superglobalne 68
Używanie $_ENV oraz $_SERVER 70
Odniesienia 71
Stałe 72
Tablice 75

6. Operatory ... 93
Operatory arytmetyczne 93
Operatory przypisania 94
Operatory łańcuchów 94
Operatory poziomu bitowego 95
Operatory porównania 96
Operatory zwiększenia i zmniejszenia o jednostkę 97
Operatory logiczne 98
Kilka przykładów operatorów 99
Operator trójkowy 100
Operator wykonania 100
Operator pierwszeństwa i asocjacyjny 101

7. Encyklopedia funkcji ... 103
Nieudokumentowane funkcje 104
Obsługa znaków nieangielskich 104

8. PHP zorientowany obiektowo 145
Ogólny opis pojęcia 145
Klasy 146
Obiekty 148
Właściwości 149
Zmienna 'this' 150
Obiekty wewnątrz obiektów 150
Modyfikatory kontroli dostępu 151
Informacja o typie obiektu 157
Wskazówki dotyczące typu klasy 158
Konstruktory i destruktory 159
Kopiowanie obiektów 162
Porównywanie obiektów za pomocą == i === 163
Zapisywanie obiektów 164
Magiczne metody 165
Statyczne metody i właściwości klas 168
Przydatne funkcje pomocnicze 169
Interfejsy 170
Dereferencja wartości zwrotnych obiektu 172

9. Formularze HTML 173
Co oznacza być dynamicznym? 174
Projektowanie formularza 174
Obsługa danych 178
Rozdzielenie formularza na wiele stron 182
Sprawdzanie poprawności danych wejściowych 183
Projektowanie formularza 185
Podsumowanie 186

10. Cookies i sesje 187
Cookies kontra sesje 187
Używanie cookies 188
Używanie sesji 190
Przechowywanie złożonych typów danych 196

11. Buforowanie danych wyjściowych 197
Dlaczego używać buforowania danych wyjściowych? 197
Rozpoczynamy 198
Ponowne używanie buforów 198
Bufory kaskadowe 199
Opróżnianie buforów kaskadowych 199
Odczytywanie buforów 200

Inne funkcje OB 201
Opróżnianie danych wyjściowych 201
Kompresja danych wyjściowych 203
Przepisywanie URL 204

12. Bezpieczeństwo 207
Podpowiedzi dotyczące bezpieczeństwa 207
Szyfrowanie 209

13. Pliki 213
Odczytywanie plików 213
Tworzenie i zmiana plików 217
Przenoszenie, kopiowanie i usuwanie plików 219
Inne funkcje dotyczące plików 220
Sprawdzenie, czy plik istnieje 221
Otrzymywanie informacji dotyczących czasu pliku 222
Szczegółowa analiza informacji w nazwie pliku 222
Obsługa przekazywania plików na serwer 223
Blokowanie plików za pomocą funkcji flock() 225
Odczytywanie praw dostępu pliku i jego statusu 226
Zmiana prawa dostępu i własności pliku 227
Praca z dowiązaniami 228
Praca z katalogami 229
Zdalne pliki 231
Sumy kontrolne plików 231
Przetwarzanie pliku konfiguracyjnego 231

14. Bazy danych 235
Używanie MySQL z PHP 235
PEAR::DB 242
SQLite 247
Stałe połączenia 251
Udoskonalony MySQL 252

15. Wyrażenia regularne 253
Podstawy regexps: preg_match() i preg_match_all() 253
Klasy znaków regexps 254
Znaki specjalne wyrażeń regularnych 254
Słowa i wolne przestrzenie wyrażeń regularnych 258
Przechowywanie dopasowanych łańcuchów 258
Zastępowanie wyrażeniami regularnymi 259
Przykłady składni wyrażeń regularnych 260
Program „Regex Coach” 261

16. Operacje na obrazkach 263
Rozpoczynamy 263
Wybór formatu 265
Rozpoczynamy tworzenie 266
Więcej kształtów 267
Złożone kształty 269
Wyświetlanie tekstu 270
Wczytywanie istniejących obrazków 273
Kolor i wypełnianie obrazka 275
Dodanie przezroczystości 277
Używanie pędzli 278
Podstawy kopiowania obrazków 280
Skalowanie i rotacja 282
Punkty i linie 285
Efekty specjalne uzyskane za pomocą funkcji imagefilter() 287
Przeplot obrazka 290
Pobieranie typu MIME obrazka 290

17. Tworzenie plików PDF 293
Rozpoczynamy 293
Dodanie większej liczby stron oraz dodatkowych stylów 295
Dodanie obrazków 296
Efekty specjalne PDF 297
Dołączenie danych dokumentu 297

18. Tworzenie klipów Flash 299
Prosty klip 299
Tekst w klipach Flash 301
Akcje 302
Animacja 304

19. XML & XSLT 307
Rozszerzenie SimpleXML 307
Transformacja XML za pomocą XSLT 313

20. Programowanie sieciowe 315
Gniazda 315
HTTP 318
Wysyłanie wiadomości e-mail 322
Curl 329

21. Rozprowadzanie swojego kodu 337
Kod niezależny od platformy 1: wczytywanie rozszerzeń 337
Kod niezależny od platformy 2: używanie rozszerzeń 338
Kod niezależny od platformy 3: ścieżki i separatory wierszy 338
Kod niezależny od platformy 4: różnice w pliku php.ini 339
Kod niezależny od platformy 5: sprawdzanie wersji PHP za pomocą funkcji phpversion() i version_compare() 340

22. Debugowanie 341
Najbardziej podstawowa technika debugowania 341
Tworzenie warunków 342
Zgłaszanie swoich błędów 344
Testowanie za pomocą funkcji php_check_syntax() 345
Kolorowanie kodu źródłowego 345
Obsługa błędów MySQL 346
Obsługa wyjątków 347
Wsteczne przeglądanie swojego kodu 350
Własna obsługa błędów 351
Własna obsługa wyjątków 354
Użycie symbolu @ do zablokowania błędów 354
Funkcja phpinfo() 355
Styl danych wyjściowych 355

23. Wydajność 359
Rozsądnie napisz swój kod 359
Użycie Zend Optimizer 360
Użycie bufora kodu PHP 360
Kompresja danych wyjściowych 360
Nie używaj CGI 361
Przeprowadź debugowanie swojego kodu 361
Używaj stałych połączeń 361
Prawidłowa kompilacja 362

Skorowidz 363

ROZDZIAŁ 4.

Język PHP

Niniejszy rozdział stanowi kompletne wprowadzenie do podstaw programowania PHP; przedstawia zmienne, komentarze, instrukcje warunkowe, pętle oraz inne konstrukcje. Jeżeli posiadasz niewielkie doświadczenie w PHP, najlepiej rozpocząć pracę właśnie od lektury tego rozdziału. W przeciwnym przypadku możesz zapoznać się z poszczególnymi jego fragmentami w celu odświeżenia pamięci.

Podstawy PHP

PHP działa domyślnie z wyłączonym trybem PHP, co oznacza, że dopóki tryb PHP nie zostanie włączony, zawartość będzie traktowana jako zwykły tekst, a nie jako kod PHP. Taka metoda analizy powoduje, że elementy skryptu PHP są „blokami kodu" — samodzielnymi fragmentami kodu, które mogą działać niezależnie od otaczającego je kodu HTML.

W celu oznaczenia typu skryptów PHP są one zwykle zapisywane z rozszerzeniem *.php*. Ilekroć Twój serwer WWW zostanie poproszony o wysłanie pliku kończącego się na *.php*, wówczas w pierwszej kolejności przekaże plik do interpretera PHP. Z kolei interpreter wykona kod zawarty w skrypcie, a następnie zwróci użytkownikowi wygenerowany plik. Podstawowa jednostka kodu PHP jest nazwana poleceniem i kończy się znakiem średnika oznaczającym zakończenie polecenia. W celu zachowania przejrzystości jeden wiersz kodu zazwyczaj zawiera jedno polecenie, choć możesz umieścić w nim tyle poleceń, ile tylko zechcesz. Poniższe dwa przykłady wykonują to samo zadanie:

```
<?php
   // opcja 1
   print "Witaj, ";
   print "świecie!";

   // opcja 2
   print "Witaj, "; print "świecie!";
?>
```

Puryści PHP wskazują, że technicznie rzecz biorąc `print` nie jest funkcją i z tego punktu widzenia mają rację. Dlatego też polecenie `print` nie wymaga nawiasów klamrowych wokół przekazywanych do niego danych. Do innych konstrukcji języka, które maskują się jako funkcje (i wspominamy tutaj o nich ze względu na zachowanie porządku), należą polecenia: `array`, `echo`, `include`, `require`, `return` i `exit`.

Możesz użyć nawiasów z tymi konstrukcjami; jest to zupełnie nieszkodliwe:

```
<?php
   print("Witaj!");
?>
```

Chociaż na pierwszy rzut oka polecenia `print` i `echo` wydają się identyczne, to jednak nie są. W przeciwieństwie do polecenia `echo`, konstrukcja `print` zachowuje się bardziej jako funkcja, ponieważ zwraca wartość (`1`). Jednakże polecenie `echo` jest bardziej użyteczne, gdyż możesz do niego przekazać wiele parametrów, na przykład w ten sposób:

```
<?php
   echo "To ", "jest ", "test.";
?>
```

Aby uzyskać taki sam efekt za pomocą konstrukcji `print`, zamiast znaku przecinka musiałbyś użyć operatora połączenia (`.`) i połączyć ze sobą łańcuchy. Jeżeli, podobnie jak w powyższym przykładzie, posiadasz kilka elementów do wyświetlenia, wówczas ze względu na zachowanie przejrzystości zalecane jest użycie polecenia `echo`.

Zmienne

Zmienne w PHP — czyli elementy, które przechowują dane — rozpoczynają się znakiem `$`, następnie występuje znak podkreślenia lub litera, a dalej kombinacja liter, cyfr i znaków podkreślenia. Oznacza to, że nie możesz rozpocząć nazwy zmiennej od cyfry. Jednym godnym uwagi wyjątkiem w ogólnym schemacie nazewnictwa zmiennych są „zmienne zmiennych", które zostaną przedstawione w kolejnym rozdziale. Lista poprawnych i niepoprawnych nazw zmiennych została przedstawiona w tabeli 4.1.

Tabela 4.1. Poprawne i niepoprawne nazwy zmiennych

`$mojazmienna`	Poprawna nazwa.
`$Imie`	Poprawna nazwa.
`$_Wiek`	Poprawna nazwa.
`$___WIEK___`	Poprawna nazwa.
`$91`	Niepoprawna nazwa, rozpoczyna się cyfrą.
`$1Imie`	Niepoprawna nazwa, rozpoczyna się cyfrą.
`$Imie91`	Poprawna nazwa, cyfry są poprawne, jeśli są na końcu i po pierwszym znaku.
`$_Imie91`	Poprawna nazwa.
`$Imie's`	Niepoprawna nazwa, nie są dozwolone inne symbole niż „_", a więc apostrof jest niedozwolony.

Zmienne rozróżniają wielkość liter, co oznacza, że zmienna `$Foo` nie jest tą samą zmienną co: `$foo`, `$FOO` lub `$fOO`.

Przypisywanie zmiennych jest tak proste jak używanie na zmiennej operatora przypisania (`=`), a następnie podanie wartości, która ma zostać przypisana. Poniżej znajduje się prosty skrypt przedstawiający przypisanie oraz dane wyjściowe. Zwróć uwagę na znaki średnika użyte na końcu każdego polecenia:

```
<?php
   $imie = "Paweł";
   print "Twoje imię to $imie\n";
   $imie2 = $imie;
   $wiek = 20;
   print "Twoje imię to $imie2, a Twój wiek to $wiek lat\n";
   print 'Do widzenia, $imie!\n';
?>
```

W powyższym przykładzie zmiennej `$imie` przypisaliśmy łańcuch `Paweł`, a PHP pozwolił nam wyświetlić tę zmienną po fragmencie „Twoje imię to”. Ponieważ PHP w każdym znalezionym przez siebie miejscu (lub wewnątrz łańcuchów ujętych w podwójny znak cudzysłowu, to znaczy rozpoczynające się i kończące na `"`) wystąpienia zmiennej `$imie` zastąpił zmienną jej wartością, dane wyjściowe z pierwszego polecenia to „Twoje imię to Paweł”.

Następnie ustawiamy zmienną `$imie2`, aby była zmienną `$imie`, co powoduje skopiowanie wartości zmiennej `$imie` do zmiennej `$imie2`. Po tej operacji zmienna `$imie2` ma wartość `Paweł`. Ustawiamy również zmienną `$wiek` i przypisujemy jej wartość `20`. Nasze drugie polecenie `print` wyświetla jednocześnie obie zmienne. I tym razem PHP zastępuje zmienne ich rzeczywistymi wartościami.

Jednakże ostatnie polecenie `print` nie zastąpi zmiennej `$imie` wartością `Paweł`. Zamiast tego zostanie wyświetlone:

```
Do widzenia, $imie!\n
```

Powodem takiego stanu rzeczy jest fakt, że PHP nie wykonuje zastąpienia zmiennej wewnątrz łańcuchów ujętych w znaki pojedynczego cudzysłowu oraz nie zastępuje większości znaków sterujących (wyjątkiem jest `\'`). W łańcuchach ujętych w podwójne znaki cudzysłowu PHP zastąpi zmienną `$imie` jej wartością, natomiast w łańcuchu ujętym w pojedyncze znaki cudzysłowu PHP interpretuje, że chcesz otrzymać tekst `$imie` jako dane wyjściowe.

Kiedy chcesz dodać cokolwiek do swojej zmiennej wewnątrz łańcucha, PHP może uznać te znaki za część zmiennej. Na przykład:

```
<?php
   $owoc = "grejpfrut";
   print "Te $owocy nie są jeszcze dojrzałe.";
?>
```

Podczas gdy pożądanymi danymi wyjściowymi było zdanie „Te grejpfruty nie są jeszcze dojrzałe.”, to jednak rzeczywiste dane wyjściowe są inne. Ponieważ dodaliśmy znak „y” na końcu nazwy zmiennej, zmieniliśmy ją z `$owoc` na `$owocy`. Zmienna `$owocy` nie istnieje, więc PHP pozostawił puste miejsce i może wygenerować błąd. Są dwa sposoby rozwiązania tego typu sytuacji:

```
<?php
   $owoc = "grejpfrut";
   print "Te ${owoc}y nie są jeszcze dojrzałe.";
   print "Te {$owoc}y nie są jeszcze dojrzałe.";
?>
```

Nawiasy { i } używane wewnątrz łańcucha technicznie oznaczają zmienną zmiennej, ale w przykładzie przedstawionym powyżej informują PHP, w którym miejscu następuje koniec zmiennej. Nie musisz używać nawiasów wówczas, gdy dodawane do zmiennej znaki powodują, że nazwa zmiennej staje się niepoprawna, na przykład w poniższej sytuacji:

```
<?php
   $owoc = "grejpfrut";
   print "Smak tego $owoc'u nie jest dobry.";
?>
```

Taki zapis będzie funkcjonował, ponieważ nie wolno używać apostrofów jako części nazwy zmiennej.

Wolne przestrzenie

Znaki odstępu, tabulatory i puste wiersze między poleceniami nie mają wpływu na sposób wykonywania kodu. Dla PHP następny przedstawiony skrypt jest traktowany jak każdy inny, niezależnie od faktu, że niektóre polecenia są w tym samym wierszu, a inne zostały rozbite na kilka wierszy:

```
<?php
   $imie = "Paweł"; print "Twoje imię to $imie\n";
   $imie2 = $imie; $wiek = 20;

   print "Twoje imię to $imie2, a Twój wiek to $wiek lat\n";

   print 'Do widzenia, $imie!\n';
?>
```

Należy używać wolnych przestrzeni do rozdzielenia Twojego kodu na czytelne bloki, tak aby ich znaczenie było łatwe do zrozumienia dzięki wizualnemu przedstawieniu rozmieszczenia.

Heredoc

Jeżeli posiadasz długi łańcuch, możesz rozważyć użycie składni heredoc. W zasadzie heredoc pozwala Ci na zdefiniowanie własnych ograniczników łańcucha, tak więc możesz użyć innych znaków niż pojedynczy lub podwójny cudzysłów. Na przykład możemy użyć łańcucha `EOT` (*end of text* — koniec tekstu) jako naszego ogranicznika. Dzięki temu pozbywamy się z głównej części tekstu znaków pojedynczego i podwójnego cudzysłowu — łańcuch zostanie zakończony jedynie po wpisaniu `EOT`.

Wydaje się to trochę bardziej skomplikowanie niż jest w praktyce. Ogranicznik łańcucha musi być jedynym elementem od samego początku wiersza. Oznacza to, że nie możesz dodać wokół niego znaków odstępu bądź tabulatorów. Poniżej jest przedstawiony działający przykład tego typu składni:

```
<?php
   $mojlancuch = <<<EOT
      To jest pewien tekst PHP.
      Tekst jest całkowicie dowolny.
      Mogę w nim użyć znaków "podwójny cudzysłów"
      oraz 'pojedynczy cudzysłów',
      a nawet $zmiennych, które zostaną prawidłowo
      zamienione na ich wartości.
      Możesz nawet wpisać łańcuch EOT, tak długo jak nie będzie on
      jedynym łańcuchem w wierszu, jak ma to miejsce poniżej:
EOT;
?>
```

Należy zwrócić uwagę na kilka ważnych informacji o heredoc oraz powyższym przykładzie:

- Możesz użyć dowolnych znaków jako ogranicznika łańcucha, `EOT` jest tylko przykładem.
- Przed ogranicznikiem musisz użyć zapisu `<<<` w celu poinformowania PHP, że chcesz wejść do trybu heredoc.
- Włączone jest zastępowanie zmiennych, co oznacza, że musisz zmienić znaczenie znaku dolara, jeśli nie chcesz, aby PHP zastąpiło zmienną jej wartością.
- Możesz użyć ogranicznika w dowolnym miejscu w tekście, z wyjątkiem pierwszej kolumny nowego wiersza.
- Na końcu łańcucha wpisz ogranicznik bez otaczających go wolnych przestrzeni, a następnie dodaj znak średnika.

Bez składni heredoc skomplikowane przypisania łańcuchów mogłyby być bardzo zawiłe.

Krótkie wprowadzenie do typów zmiennych

Zmienne w PHP mogą być typu: całkowitego (liczby całkowite), zmiennoprzecinkowego (zwykle nazywane „float"; są to liczby zmiennoprzecinkowe), łańcuch (zestaw znaków), tablica (grupa danych), obiekt (kompleksowe połączenie danych i funkcjonalności) lub zasób (jakiekolwiek zewnętrzne dane, na przykład obrazek). Poszczególne typy poznamy bardziej szczegółowo w dalszej części książki. W chwili obecnej musimy jedynie wiedzieć, czym są zmienne i jak one działają.

Bloki kodu

PHP dość intensywnie korzysta z bloków kodu — porcji kodu PHP, który jest oddzielony od pozostałej części skryptu. W trakcie czytania dalszych podrozdziałów tego rozdziału zauważysz, że PHP używa nawiasów klamrowych `{` i `}` do otwarcia i zamknięcia bloków kodu.

Otwarcie i zamknięcie bloków kodu

Istnieje wiele sposobów na otwarcie bloku kodu PHP (przejście do trybu analizy składniowej PHP); wybierz ten, który odpowiada Ci najbardziej. Zalecanym sposobem jest użycie znacznika `<?php` w celu wejścia do trybu PHP oraz `?>` do opuszczenia trybu PHP, choć możesz również skorzystać ze skróconych wersji znaczników, `<?` i `?>`.

Skrócony zapis ma jedną dużą zaletę oraz dwie poważne wady: możesz otrzymać dane wyjściowe ze swojego skryptu dzięki użyciu specjalnego zapisu `<?=`, jak ma to miejsce w poniższym przykładzie:

```
<?="Witaj, świecie!" ?>
```

Poniżej znajduje się odpowiednik przedstawionego wyżej skryptu, tym razem napisany przy użyciu standardowych otwierających i zamykających znaczników PHP:

```
<?php
   print "Witaj, świecie!";
?>
```

Skrócona wersja znaczników jest zatem bardziej zwięzła, ale trochę trudniejsza do czytania. Jednakże pierwszą wadą skróconego zapisu znaczników PHP jest fakt, że nakładają się z XML (i przez to także z XHTML), które również używają zapisu `<?` do otwarcia bloku kodu. Oczywiście, oznacza to, że jeśli będziesz próbował użyć razem XML oraz skróconych znaczników PHP, wówczas napotkasz problemy. Mamy więc pierwszy powód, dla którego zaleca się używanie standardowych otwierających i zamykających znaczników PHP. Skrócone znaczniki są zawsze niebezpieczne, ponieważ mogą zostać zablokowane w pliku konfiguracyjnym PHP, *php.ini*. Prowadzi to do tego, że Twoje skrypty nie będą przenośne.

Istnieją również dwa inne, rzadziej używane warianty. Pierwszy z nich, `<% %>`, otwiera i zamyka blok kodu w taki sam sposób, w jaki robi to Microsoft ASP. Drugim zapisem jest `<script language="php"></script>`. Te dwa warianty często sprawują się lepiej w graficznych edytorach WWW, takich jak Macromedia Dreamweaver lub Microsoft FrontPage. Nie są jednak zalecane do ogólnego użycia, ponieważ muszą zostać włączone, aby mogły funkcjonować prawidłowo.

Możesz przejść do trybu PHP oraz z niego wyjść używając znaczników `<?php` i `?>` — gdziekolwiek i jak często zechcesz.

Komentarze

Znajdując się w trybie PHP możesz zaznaczyć określone części kodu jako komentarz, a te fragmenty nie powinny zostać wykonane. Mamy trzy sposoby oznaczenia tekstu jako komentarz: `//`, `/* */` oraz #. Zapisy `//` i # oznaczają: „Zignoruj pozostałą część wiersza", podczas gdy zapis `/*` oznacza: „Zignoruj wszystko, dopóki nie natkniesz się na `*/`". W trakcie stosowania zapisu `/* */` występują pewne komplikacje, co powoduje, że ten sposób jest mniej pożądany w użyciu.

```
<?php
   print "To zostanie wyświetlone\n";
   //  print "To nie zostanie wyświetlone\n";
   #  print "To nie zostanie wyświetlone\n";
   print "To zostanie wyświetlone\n";
   /*  print "To nie zostanie wyświetlone\n";
   print "To nie zostanie wyświetlone\n";  */
?>
```

Powyższy fragment kodu przedstawia w działaniu wszystkie trzy typu komentarzy, ale nie pokazuje problemu związanego ze stylem komentowania `/* */`. Jeżeli rozpocząłeś zapisem `/*` komentarz w jednym wierszu, a zakończyłeś go znacznie niżej, gdzie rozpoczął się inny komentarz stylu `/*`, wówczas stwierdzisz, że skrypt nie działa. Jest to spowodowane faktem, że nie możesz gromadzić lub „zagnieżdżać" komentarzy stylu `/* */`. Próba takiego działania zakończy się spektakularnym niepowodzeniem.

Najlepszym wyjściem w celu komentowania jest zastosowanie komentarzy typu `//`, ponieważ są one proste, łatwe do zlokalizowania, odczytania i kontrolowania.

Instrukcje warunkowe

PHP pozwala na wybór podejmowanej akcji, w zależności od wyników spełnienia warunku. Warunek może zostać dowolnie przez Ciebie wybrany; dysponujesz również możliwością łą-

czenia warunków, co pozwala na tworzenie bardziej skomplikowanych akcji. Poniżej znajduje się działający przykład:

```
<?php
   $Wiek = 20;
   if ($Wiek < 18) {
      print "Jesteś zbyt młody - baw się dobrze!\n";
   } else {
      print "Nie masz poniżej 18 lat.\n";
   }

   if ($Wiek >= 18 && $Wiek < 50) {
      print "Jesteś w najlepszym okresie swojego życia!\n";
   } else {
      print "Nie jesteś w najlepszym okresie swojego życia.\n";
   }

   if ($Wiek >= 50) {
      print "Będziesz mógł wkrótce odpocząć - hurra!\n";
   } else {
      print "Nie będziesz mógł wkrótce odpocząć :-( ";
   }
?>
```

Na najbardziej podstawowym poziomie PHP ocenia polecenie `if` od lewej do prawej strony, co oznacza, że w pierwszej kolejności jest sprawdzane, czy zmienna `$Wiek` jest równa bądź większa od `18`. Dopiero w dalszej kolejności następuje sprawdzenie, czy zmienna `$Wiek` jest mniejsza od `50`. Podwójny znak `&&` oznacza, że oba wyrażenia muszą przyjąć wartość `true`, aby został wykonany fragment kodu `print "Jesteś w najlepszym okresie swojego życia\n"`. Jeżeli z jakiegokolwiek powodu jeden z warunków nie przyjmuje wartości `true`, wówczas zostanie wyświetlone zdanie: „Nie jesteś w najlepszym okresie swojego życia”. Kolejność, w której są sprawdzane warunki, jest uzależniona od pierwszeństwa operatora. Zagadnienia związane z pierwszeństwem operatorów zostaną omówione w następnym rozdziale.

Oprócz operatora `&&` występuje również operator `||` (podwójna pionowa linia), który oznacza `OR` (lub). W takiej sytuacji całe wyrażenie przyjmuje wartość `true`, jeżeli jeden z warunków przyjmie wartość `true`.

Mamy kilka sposobów na porównanie dwóch liczb. Do tej pory poznaliśmy `<` (mniejszy niż), `<=` (mniejszy bądź równy) i `>=` (większy bądź równy). Pełną listę zaprezentujemy dalej, ale w pierwszej kolejności należy wspomnieć o jednym ważnym operatorze: `==` (dwa znaki równości obok siebie). Oznaczają one „jest równy”. Dlatego też warunek `1 == 1` jest prawdziwy, natomiast `1 == 2` nie jest prawdziwy.

Kod przeznaczony do wykonania w instrukcjach warunkowych `if` jest umieszczony w swoim własnym bloku kodu (pamiętaj, blok kodu rozpoczyna się nawiasem klamrowym `{`, a kończy nawiasem klamrowym `}`). Natomiast kod przeznaczony do wykonania w przypadku, gdy warunek nie zostanie spełniony, znajduje się w bloku `else`. Dzięki takiemu układowi PHP nie będzie próbował wykonać kodu zarówno wtedy, gdy wyrażenie przyjmuje zarówno wartość `true`, jak i `false`.

Jedną kwestią wartą odnotowania jest fakt, że PHP praktykuje „jak najkrótsze polecenie if” — będzie próbował wykonać minimalną dopuszczalną liczbę warunków. W zasadzie przestanie sprawdzać warunki, gdy tylko upewni się, że może to zrobić. Na przykład:

```
if ($Wiek > 10 && $Wiek < 20)
```

Jeżeli zmienna `$Wiek` przyjmie wartość `8`, wówczas pierwszy warunek (`$Wiek > 10`) przyjmie wartość `false`. PHP nie będzie więc sprawdzać drugiego warunku. Oznacza to, na przykład, że możesz sprawdzić, czy zmienna została ustalona i czy posiada określoną wartość. Jeżeli zmienna nie została ustalona, PHP skróci instrukcję `if` i nie będzie sprawdzać jej wartości. Jest to prawidłowe działanie, ponieważ jeśli będziesz sprawdzać wartość nieustalonej zmiennej, wówczas PHP zgłosi błąd.

Pomocnym dodatkiem do instrukcji `if` jest polecenie `elseif`, które pozwala na łączne sprawdzenie warunków w bardziej inteligentny sposób:

```
<?php
   if ($Wiek < 10) {
      print "Masz poniżej 10 lat";
   } elseif ($Wiek < 20) {
      print "Masz poniżej 20 lat";
   } elseif ($Wiek < 30) {
      print "Masz poniżej 30 lat";
   } elseif ($Wiek < 40) {
      print "Masz poniżej 40 lat";
   } else {
      print "Masz ponad 40 lat";
   }
?>
```



Użytkownicy Perla powinni zwrócić uwagę, że piszemy `elseif`, a nie `elsif`.

Ten sam efekt możesz uzyskać za pomocą poleceń `if`, ale użycie `elseif` jest łatwiejsze do odczytania. Ujemną stroną takiego rozwiązania jest to, że zmienna `$Wiek` musi zostać ponownie sprawdzona.

Jeżeli posiadasz tylko jedno polecenie kodu do wykonania, możesz się zupełnie obejść bez nawiasów klamrowych. Taki wariant jest bardzo czytelny.

Poniżej są dwa fragmenty kodu, których wynik jest identyczny:

```
if ($zablokowany) {
   print "Nie masz wstępu!";
}

if ($zablokowany) print "Nie masz wstępu!";
```

Case Switching

Twoje bloki kodu `if...elseif` mogą osiągnąć duże rozmiary, kiedy będziesz sprawdzał wiele warunków względem tej samej zmiennej, jak zostało to przedstawione poniżej:

```
<?php
   $imie = "Bartek";
   if ($imie == "Janek") {
      print "Twoje imię to Janek\n";
   } elseif ($imie == "Lidia") {
      print "Twoje imię to Lidia\n";
   } elseif ($imie == "Bartek") {
      print "Twoje imię to Bartek\n";
   } elseif ($imie == "Stasia") {
      print "Twoje imię to Stasia\n";
```

```
    } else {
        print "Nie wiem, jak masz na imię!\n";
    }
?>
```

PHP posiada odpowiednie rozwiązanie dla takich sytuacji: `switch/case`. W bloku instrukcji `switch/case` określasz sprawdzany warunek oraz podajesz listę możliwych wartości, które chcesz obsłużyć. Używając poleceń `switch/case` możemy przepisać w następujący sposób nasz poprzedni skrypt:

```
<?php
    $imie = 'Bartek';
    switch($imie) {
    case "Janek":
        print "Twoje imię to Janek\n";
        break;
    case "Lidia":
        print "Twoje imię to Lidia\n";
        break;
    case "Bartek":
        print "Twoje imię to Bartek\n";
        break;
    case "Stasia":
        print "Twoje imię to Stasia\n";
        break;
    default:
        print "Nie wiem, jak masz na imię!\n";
    }
?>
```

Polecenia `switch/case` są często używane do sprawdzenia różnego rodzaju danych i zabierają znacznie mniej miejsca niż odpowiadające im polecenia `if`.

Mamy dwie istotne kwestie warte odnotowania w kodzie poleceń `switch/case` PHP. Po pierwsze, przed zapisem „default" nie umieszczamy słowa `case` — jest to po prostu sposób działania języka. Po drugie, każdą z akcji `case` kończymy poleceniem „break;". Jest to spowodowane faktem, że kiedy PHP odnajdzie na liście pasujący warunek, wykona przypisaną do niego akcję oraz akcje wszystkich pozostałych odpowiadających warunków umieszczonych poniżej (na ekranie przedstawionych na dole). Taki sposób działania został zaczerpnięty bezpośrednio z języka C i w zasadzie jest sprzeczny z intuicją oraz naszym sposobem myślenia. Rzadko się zdarza sytuacja, w której chciałbyś wykluczyć polecenie `break` z końca swoich warunków `case`.

Warunek `default` zostanie wykonany, jeżeli PHP nie znajdzie odpowiadającej wartości w innych warunkach lub jeśli został wykonany poprzedzający go warunek, który nie posiadał polecenia `break`.

Słowo kluczowe „break" oznacza „Wyjdź z polecenia switch/case", a efektem jego działania jest zatrzymanie wykonywania przez PHP akcji z wszystkich warunków `case`, które znajdują się po znalezieniu tego pasującego. Bez tego polecenia nasz skrypt spowodowałby wyświetlenie:

```
Twoje imię to Bartek
Twoje imię to Stasia
Nie wiem, jak masz na imię!
```

Pętle

PHP posiada następujące słowa kluczowe dotyczące pętli: `foreach`, `while`, `for` oraz `do...while`.

Pętla `foreach` została zaprojektowana do pracy z tablicami, a jej działanie polega na kolejnym przejściu przez każdy element w tablicy. Możesz jej również użyć z obiektami; w takim przypadku przechodzi kolejno przez każdą publiczną zmienną tego obiektu.

Najbardziej podstawowe użycie pętli `foreach` wyciąga jedynie wartości z każdego elementu tablicy, jak to zostało przedstawione poniżej:

```
foreach($tablica as $wartosc) {
   print $wartosc;
}
```

W zaprezentowanym przykładzie tablica `$tablica` zostaje poddana działaniu pętli, a jej wartości zostają wyciągnięte do zmiennej `$wartosc`. W takiej sytuacji klucze tablicy są całkowicie ignorowane, co zwykle jest bardziej sensowne, jeśli zostają wygenerowane automatycznie (na przykład 0, 1, 2, 3 itd.).

Oczywiście, pętli `foreach` możesz również użyć do wydobycia kluczy, na przykład w taki sposób:

```
foreach($tablica as $klucz => $wartosc) {
   print "$klucz = $wartosc\n";
}
```

W trakcie pracy z obiektami składnia jest identyczna:

```
<?php
   class monitor {
      private $Marka;
      public $Wielkosc;
      public $Rozdzielczosc;
      public $CzyJestPlaski;

      public function __construct($Marka, $Wielkosc, $Rozdzielczosc, $CzyJestPlaski) {
         $this->Marka = $Marka;
         $this->Wielkosc = $Wielkosc;
         $this->Rozdzielczosc = $Rozdzielczosc;
         $this->CzyJestPlaski = $CzyJestPlaski;
      }
   }

   $AppleCinema = new monitor("Apple", "30", "2560x1600", true);

   foreach($AppleCinema as $zmienna => $wartosc) {
      print "$zmienna = $wartosc\n";
   }
?>
```

Pętle PHP `while` są używane do wykonywania bloku kodu dopóty, dopóki warunek jest prawdziwy. Na przykład poniższy kod wykona pętlę od 1 do 10, wyświetlając aktualną wartość:

```
<?php
   $i = 1;
   while($i <= 10) {
      print "Liczba $i\n";
      $i = $i + 1;
   }
?>
```

Zwróć uwagę, że ponownie PHP używa bloków kodu przedstawiających zakres naszej pętli — pętle `while` rozpoczynają się znacznikiem otwierającym (`{`), a kończą znacznikiem zamykającym (`}`) — w celu wyraźnego „powiedzenia" PHP, które wiersze kodu powinny zostać uwzględnione w pętli.

Podobnie jak w przypadku polecenia `if`, możesz umieścić jakiekolwiek wybrane przez siebie warunki w pętlach `while`. Istotną kwestią pozostaje zmiana wartości warunku po każdej pętli; w przeciwnym wypadku pętla będzie wykonywana w nieskończoność.

Pętle `while` są najczęściej używane do zwiększenia o jednostkę listy, gdy nie ma ograniczenia dotyczącego liczby wykonań pętli. Na przykład:

```
while(wciąż są wiersze do odczytania z bazy danych) {
   odczytaj wiersz;
   przejdź do kolejnego wiersza;
}
```

Bardziej rozpowszechnioną formą pętli jest pętla `for`, która jest nieco bardziej skomplikowana. Pętla `for` jest tworzona na podstawie deklaracji, warunku oraz akcji. Deklaracją jest zdefiniowanie zmiennej — licznika pętli oraz ustawienie jej wartości początkowej. Warunkiem jest sprawdzenie zmiennej — licznika pętli w stosunku do wartości. Natomiast akcja to działanie, które powinno nastąpić po każdorazowej iteracji do zmiany licznika pętli.

Poniżej znajduje się przykład pętli `for` w PHP:

```
<?php
   for ($i = 1; $i < 10; $i++) {
      print "Liczba $i\n";
   }
?>
```

Jak możesz zobaczyć, pętla for posiada trzy części oddzielone średnikami. W części deklaracyjnej ustalamy zmienną `$i`, której przypisujemy wartość `1`. W części zawierającej warunek wykonujemy pętlę, jeżeli zmienna `$i` ma wartość mniejszą niż `10`. Na końcu, jako akcję, wykonujemy dodawanie cyfry `1` do wartości zmiennej `$i` w trakcie każdego wykonania pętli. Dlatego też za każdym razem jest wykonywany kod pętli.

Kiedy skrypt zostanie uruchomiony, będzie liczył od 1 do 10, wyświetlając przy tym aktualną liczbę. Zwróć uwagę, że w rzeczywistości nie zostanie wyświetlony tekst „Liczba 10", ponieważ określiliśmy, że zmienna `$i` musi być *mniejsza* od 10, nie zaś mniejsza bądź równa. Poniżej znajdują się dane wyjściowe omawianego skryptu:

```
Liczba 1
Liczba 2
Liczba 3
Liczba 4
Liczba 5
Liczba 6
Liczba 7
Liczba 8
Liczba 9
```

Konstrukcja `do...while` jest w PHP podobna do pętli `while`. Różnicą jest to, że pętla `do...while` zostanie wykonana przynajmniej jeden raz. Spróbuj przeanalizować poniższy fragment kodu:

```
<?php
   $i = 11;
   do {
```

```
        print "Liczba $i\n";
    } while ($i < 10);
?>
```

Jeśli zostanie użyty powyższy kod, tekst „Liczba 11" zostanie wyświetlony przed porównaniem zmiennej `$i` względem liczby `10`. Jeżeli w trakcie sprawdzenia zmienna `$i` jest mniejsza od 10, wówczas pętla zostanie wykonana ponownie. Dla porównania, ten sam kod może zostać napisany przy użyciu pętli `while`:

```
<?php
    $i = 11;
    while ($i < 10) {
        print "Liczba $i\n";
?>
```

Różnica jest taka, że pętla `while` mogłaby nie przedstawić żadnych danych wyjściowych, ponieważ sprawdza wartość zmiennej `$i`, zanim rozpocznie wykonywanie pętli. Dlatego też pętle `do...while` są zawsze wykonywane przynajmniej jeden raz.

Nieskończone pętle

Prawdopodobnie uznasz to za zaskakujące, ale nieskończone pętle mogą być często pomocne w Twoich skryptach. Jeżeli piszesz program, który pozwala użytkownikom na wpisywanie danych tak długo, jak oni zechcą, to nie będzie on działał, jeśli umieścisz skrypt w pętli wykonywanej 30000 razy lub nawet 30000000 razy. Zamiast tego kod powinien zostać zapętlony w nieskończoność, bezustannie przyjmując dane wprowadzane przez użytkownika, dopóki nie zakończy on programu poprzez naciśnięcie kombinacji *Ctrl+C*.

Poniżej znajdują się dwa typy najczęstszych pętli działających w nieskończoność:

```
<?php
    while(1) {
        print "W pętli!\n";
    }
?>
```

Ponieważ wartość „1" również przyjmuje wartość prawda, wykonywanie pętli będzie trwało w nieskończoność.

```
<?php
    for (;;) {
        print "W pętli!\n";
    }
?>
```

W tym przykładzie, w pętli `for` brakuje części określających deklarację, warunek i akcję, co oznacza, że pętla będzie wykonywana w nieskończoność.

Specjalne słowa kluczowe pętli

PHP dostarcza Ci słów kluczowych `break` i `continue`, służących do kontrolowania operacji pętli. W jednym z poprzednich przykładów używaliśmy już polecenia `break` podczas omawiania poleceń `switch/case`. Wówczas break było wykorzystywane do wyjścia z bloku `switch/case`, taki sam efekt polecenie wywołuje w pętli. Kiedy używamy go wewnątrz pętli do kie-

rowania zachowaniem pętli, polecenie `break` powoduje, że PHP opuszcza pętlę i kontynuuje działanie natychmiast po niej. Polecenie `continue` zmusza PHP do przeskoczenia pozostałej części bieżącej iteracji pętli i przejścia do kolejnej.



Użytkownicy Perla powinni zwrócić uwagę na to, że polecenia `break` i `continue` są odpowiednikami poleceń `last` i `next` w Perlu.

Na przykład:

```
<?php
   for ($i = 1; $i < 10; $i = $i + 1) {
      if ($i == 3) continue;
      if ($i == 7) break;
      print "Liczba $i\n";
   }
?>
```

Jest to zmodyfikowana wersja naszego oryginalnego skryptu dla pętli `for`. Tym razem dane wyjściowe przedstawiają się następująco:

```
Liczba 1
Liczba 2
Liczba 4
Liczba 5
Liczba 6
```

Zauważ, że brakuje tekstu „Liczba 3", a wykonywanie skryptu zostaje zakończone po wyświetleniu „Liczba 6". Kiedy bieżącą liczbą jest `3`, polecenie `continue` zostaje użyte do przeskoczenia pozostałej części iteracji pętli i następuje przejście do „Liczba 4". Podobnie jeżeli aktualną liczbą jest `7`, wówczas polecenie `break` powoduje zakończenie działania pętli.

Pętle wewnątrz pętli

W PHP możesz zagnieżdżać pętle, na przykład w taki sposób:

```
for ($i = 1; $i < 3; $i = $i + 1) {
   for ($j = 1; $j < 3; $j = $j + 1) {
      for ($k = 1; $k < 3; $k = $k + 1) {
         print "I: $i, J: $j, K: $k\n";
      }
   }
}
```

Wynikiem działania powyższego kodu są następujące dane wyjściowe:

```
I: 1, J: 1, K: 1
I: 1, J: 1, K: 2
I: 1, J: 2, K: 1
I: 1, J: 2, K: 2
I: 2, J: 1, K: 1
I: 2, J: 1, K: 2
I: 2, J: 2, K: 1
I: 2, J: 2, K: 2
```

W takiej sytuacji użycie polecenia `break` jest odrobinę bardziej skomplikowane, ponieważ spowoduje ono wyjście jedynie z pętli zawierającej polecenie `break`. Na przykład:

```
for ($i = 1; $i < 3; $i = $i + 1) {
   for ($j = 1; $j < 3; $j = $j + 1) {
      for ($k = 1; $k < 3; $k = $k + 1) {
         print "I: $i, J: $j, K: $k\n";
         break;
      }
   }
}
```

Tym razem skrypt wyświetli następujące dane wyjściowe:

```
I: 1, J: 1, K: 1
I: 1, J: 2, K: 1
I: 2, J: 1, K: 1
I: 2, J: 2, K: 1
```

Jak możesz się przekonać, pętla `$k` zostanie wykonana tylko jeden raz z powodu wywołania polecenia `break`. Jednakże pozostałe pętle zostaną wykonane wiele razy. Możesz zyskać więcej kontroli przez podanie liczby po poleceniu `break`, na przykład `break 2`, co spowoduje przerwanie dwóch pętli bądź poleceń `case/switch`. Poniżej znajduje się przykład użycia polecenia `break 2`:

```
for ($i = 1; $i < 3; $i = $i + 1) {
   for ($j = 1; $j < 3; $j = $j + 1) {
      for ($k = 1; $k < 3; $k = $k + 1) {
         print "I: $i, J: $j, K: $k\n";
         break 2;
      }
   }
}
```

Wynik działania powyższego skryptu przedstawia się następująco:

```
I: 1, J: 1, K: 1
I: 2, J: 1, K: 1
```

W powyższym skrypcie pętla została wykonana jedynie dwa razy, ponieważ pętla `$k` wywołała polecenie `break 2`, które przerwało wykonanie pętli `$k` oraz `$j`. Tak więc jedynie pętla `$i` została wykonana ponownie. Moglibyśmy użyć nawet polecenia `break 3`, co oznaczałoby przerwanie wszystkich trzech pętli i powrót do normalnej kontynuacji wykonywania kodu.

Polecenie `break` ma zastosowanie zarówno w stosunku do pętli, jak i instrukcji `switch/case`. Na przykład:

```
for ($i = 1; $i < 3; $i = $i + 1) {
   for ($j = 1; $j < 3; $j = $j + 1) {
      for ($k = 1; $k < 3; $k = $k + 1) {
         switch($k) {
            case 1:
               print "I: $i, J: $j, K: $k\n";
               break 2;
            case 2:
               print "I: $i, J: $j, K: $k\n";
               break 3;
         }
      }
   }
}
```

Wiersz zawierający polecenie `break 2;` przerwie wykonywanie bloku kodu `switch/case` i pętli `$k`, podczas gdy wiersz z poleceniem `break 3;`, oprócz tych dwóch elementów, dodatkowo przerwie wykonywanie pętli `$j`. Aby przerwać wszystkie pętle z wewnątrz konstrukcji `switch/case`, wymagane jest użycie polecenia `break 4`.

Przetwarzanie trybów mieszanych

Kluczowa koncepcja w PHP daje możliwość włączenia trybu analizy składniowej PHP w dowolnym miejscu oraz dowolną liczbę razy, nawet wewnątrz bloku kodu. Poniżej znajduje się podstawowy skrypt PHP:

```
<?php
   if ($zalogowany == true) {
      print "To jest miejsce na wiele różnych elementów";
      print "To jest miejsce na wiele różnych elementów";
      print "To jest miejsce na wiele różnych elementów";
      print "To jest miejsce na wiele różnych elementów";
      print "To jest miejsce na wiele różnych elementów";
   }
?>
```

Jak możesz zobaczyć, skrypt zawiera wiele poleceń `print`, które zostaną wykonane jedynie wtedy, gdy zmienna `$zalogowany` przyjmie wartość `true`. Wszystkie dane wyjściowe zostały zahermetyzowane w poleceniach `print`, choć PHP pozwala na opuszczenie bloku kodu PHP, podczas gdy wciąż jest otwarty blok kodu polecenia `if`. Oto przykład, jak to się przedstawia w praktyce:

```
<?php
   if ($zalogowany == true) {
}
      To jest miejsce na wiele różnych elementów
      To jest miejsce na wiele różnych elementów
      To jest miejsce na wiele różnych elementów
      To jest miejsce na wiele różnych elementów
      To jest miejsce na wiele różnych elementów
<?php
   }
?>
```

Wiersze zawierające tekst „To jest miejsce na wiele różnych elementów” są wciąż wysyłane jedynie wtedy, gdy zmienna `$zalogowany` przyjmuje wartość `true`. Wychodzimy jednak z trybu PHP, aby je wyświetlić. Następnie ponownie wchodzimy do trybu PHP w celu zamknięcia polecenia `if` i kontynuacji wykonywania kodu. Dzięki takiemu podejściu cały skrypt staje się łatwiejszy w czytaniu.

Dołączanie innych plików

Jedną z najczęstszych operacji w PHP jest włączanie jednego skryptu do drugiego i współdzielenie tym samym funkcjonalności. Do włączenia jednego skryptu do innego używamy słowa kluczowego `include`, podając nazwę pliku, który chcemy włączyć.

Rozpatrzmy na przykład następujący plik — *foo.php*:

```
<?php
   print "Rozpoczynamy foo\n";
   include 'bar.php';
   print "Kończymy foo\n";
?>
```

A oto zawartość pliku *bar.php*:

```
<?php
   print "W barze\n";
?>
```

PHP wczyta plik *bar.php*, odczyta jego zawartość, a następnie umieści ją w pliku *foo.php* w miejscu wiersza `include 'bar.php'`. Dlatego też plik *foo.php* będzie przedstawiał się następująco:

```
<?php
   print "Rozpoczynamy foo\n";
   print "W barze\n";
   print "Kończymy foo\n";
?>
```

Być może dziwisz się, dlaczego został dodany jedynie wiersz zawierający tekst „W barze", a pominięte wiersze zawierające znaczniki: otwierający i zamykający. Jest to spowodowane faktem, że PHP opuszcza tryb PHP za każdym razem, gdy dodaje inny plik, a następnie po „powrocie" z pliku ponownie przechodzi do trybu PHP. Dlatego też plik *foo.php* po połączeniu z plikiem *bar.php* w rzeczywistości wygląda następująco:

```
<?php
   print "Rozpoczynamy foo\n";
?>
<?php
   print "W barze\n";
?>
<?php
   print "Kończymy foo\n";
?>
```

PHP dołącza plik jedynie wtedy, gdy wiersz zawierający polecenie `include` jest faktycznie wykonywany. Zatem poniższy kod nigdy nie dołączy pliku *bar.php*:

```
<?php
   if (53 > 99) {
      include 'bar.php';
   }
?>
```

Jeżeli spróbujesz dołączyć plik, który nie istnieje, wówczas PHP wygeneruje komunikat ostrzeżenia. Jeśli Twój skrypt koniecznie wymaga określonego pliku, PHP posiada również słowo kluczowe `require`. W przypadku użycia `require` wywołanie nieistniejącego pliku spowoduje wystąpienie błędu krytycznego i tym samym zatrzymanie wykonywania skryptu. W sytuacji gdy dołączany do skryptu plik jest niezbędny, zazwyczaj najlepszym wyjściem jest skorzystanie z polecenia `require`.



We wcześniejszych wersjach PHP polecenie `require` było odpowiednikiem bezwarunkowego polecenia `include`. Jeżeli polecenie `require` zostało umieszczone wewnątrz instrukcji warunkowej, wówczas plik był dołączany, nawet jeśli wyrażenie warunkowe przyjmowało wartość `false`. Taka sytuacja nie ma miejsca w PHP5: pliki są dołączane jedynie wtedy, gdy wyrażenie warunkowe (jeśli takie istnieje) przyjmuje wartość `true`.

Najczęstszym przykładem użycia dołączanych plików jest przechowywanie w nich powszechnie używanych funkcji, definicji obiektów oraz kodu określającego układ programu. Jeżeli na przykład Twoja witryna używa tego samego nagłówka HTML na każdej stronie, wówczas każdą stronę możesz rozpocząć od poniższego wiersza:

```
include 'header.php';
```

W ten sposób, jeśli będziesz chciał zmienić nagłówek swojej witryny, będziesz musiał przeprowadzić jedynie edycję pliku *header.php*. Dwa dodatkowe słowa kluczowe, które prawdopodobnie również będą używane, to `include_once` i `require_once`. Ich działanie odpowiada odpowiednio poleceniom `include` i `require` z tą różnicą, że plik będą dołączać tylko jeden

raz, nawet jeśli spróbujesz to zrobić wielokrotnie. Polecenia `include_once` i `require_once` współdzielą tę samą listę „dołączonych już" plików. Warto odnotować, że systemy operacyjne, które rozróżniają wielkość plików, na przykład Unix, są w stanie wielokrotnie użyć na pliku poleceń `include_once/require_once`, jeśli programista użył różnej wielkości liter w jego nazwie. Na przykład:

```
<?php
   include_once 'bar.php';
   include_once 'BAR.php';
   include_once 'Bar.php';
?>
```

W systemie Unix powyższy kod spowoduje próbę dołączenia trzech różnych plików, ponieważ Unix rozróżnia wielkość znaków. Rozwiązanie jest proste: w przypadku nazw plików należy zawsze używać małych liter. W komputerach z systemem Windows nazwy plików przeznaczonych do włączenia nie rozróżniają wielkości liter w PHP 5. Oznacza to, że dołączenie plików *BAR.php* i *bar.php* spowoduje włączenie tego samego pliku.

Kiedy próbujesz dodać plik za pomocą polecenia `include` bądź `require`, PHP w pierwszej kolejności sprawdza katalog, z którego został uruchomiony skrypt. Jeżeli plik nie zostanie znaleziony w tym katalogu, wówczas PHP sprawdza ogólną ścieżkę dostępu, która została zdefiniowana w Twoim pliku konfiguracyjnym *php.ini* przy użyciu dyrektywy `include_path`.



Za każdym razem, gdy dołączasz plik za pomocą polecenia `include` lub `require`, PHP wymaga jego kompilacji. Jeżeli korzystasz z buforowania kodu, ten problem nie występuje. W przeciwnym przypadku PHP rzeczywiście wielokrotnie przeprowadza kompilację tego samego pliku. Z tego powodu, jeśli dołączasz różne pliki w tym samym skrypcie, muszą one zostać przetworzone i skompilowane za każdym razem. Dlatego też najlepiej jest użyć funkcji `include_once()`. Jeżeli jej działanie zakończy się niepowodzeniem, spróbuj funkcji `get_included_files()` i `get_required_files()`, które informują Cię o nazwach dołączonych już plików. Wewnętrznie są one takimi samymi funkcjami, możesz więc użyć dowolnej.

Funkcje

Mimo że PHP jest dostarczane z licznymi funkcjami przeznaczonymi do wykonywania szerokiego zakresu zadań, być może zechcesz utworzyć swoje własne funkcje, gdy zajdzie taka konieczność. Jeśli zauważysz, że powtarzasz wykonywanie tych samych czynności lub będziesz chciał dzielić kod między poszczególnymi projektami, wówczas funkcje użytkownika są rozwiązaniem dla Ciebie.

Pisanie programów monolitycznych, w których wykonywanie kodu rozpoczyna się na początku programu i jest kontynuowane bez przerwy do samego końca, jest z punktu widzenia obsługi kodu uznawane jako błędne. W takiej sytuacji nie możesz ponownie użyć kodu. Dzięki tworzeniu funkcji Twój kod jest znacznie krótszy, łatwy do kontrolowania i obsługi oraz mniej podatny na wystąpienie błędów.

Prosta funkcja użytkownika

Tworzonym przez siebie funkcjom możesz nadać dowolne nazwy; obowiązują tutaj te zasady, jakie mamy w nazewnictwie zmiennych PHP (z wyjątkiem znaku `$`). Nie możesz ponownie definiować wbudowanych funkcji PHP oraz powinieneś zachować szczególną ostrożność

i upewnić się, że nazwy Twoich funkcji nie kolidują z istniejącymi funkcjami PHP. To, że Ty nie posiadasz dostępnej funkcji `imagepng()`, nie oznacza, że inni również jej nie posiadają.

Najprostsza funkcja użytkownika może wyglądać jak na poniższym przykładzie:

```
function foo() {
   return 1;
}

print foo();
```

Swoją funkcję definiujesz za pomocą słowa kluczowego `function`, a następnie nazwy funkcji oraz pary nawiasów. Rzeczywisty kod Twojej funkcji, który zostanie wykonany, znajduje się między nawiasami klamrowymi. W przypadku naszej funkcji `foo()` składa się z pojedynczego wiersza kodu `return 1;`, do którego wrócimy w dalszych wywodach.

Po zdefiniowaniu funkcji możemy traktować `foo()` jak każdą inną funkcję. Przekonaliśmy się o tym w czwartym wierszu, w którym wyświetliliśmy zwracaną przez funkcję wartość (znaną jako *wartości zwrotne*).

Wartości zwrotne

Wolno zwracać jedną (i tylko jedną) wartość zwrotną z funkcji. Do tego celu jest wykorzystywane polecenie `return`. W naszym przykładzie moglibyśmy użyć „`return 'foo';`” lub „`return 10 + 10;`” do przekazania wartości zwrotnych, ale `return 1;` jest najłatwiejszym i zwykle najczęstszym podobnie jak `return true;`.

Możesz zwrócić dowolnie wybraną zmienną, tak długo jak będzie to po prostu jedna zmienna dowolnego typu: liczba całkowita, łańcuch, połączenie z bazą danych itd. Słowo kluczowe `return` ustala wartość zwrotną funkcji jako zmienną, której możesz użyć, a następnie natychmiast wychodzi z funkcji. Możesz również użyć zapisu `return;` oznaczającego „wyjdź bez wysyłania z powrotem jakichkolwiek wartości”. Jeżeli spróbujesz przypisać zmiennej zwracaną wartość funkcji, która nie posiada zwracanej wartości (na przykład używa zapisu `return;` zamiast `return $wartosc;`), wówczas Twoja zmienna zostanie ustalona jako `NULL`.

Przyjrzyj się poniższemu skryptowi:

```
<?php
function foo() {
   print "W funkcji";
   return 1;
   print "Opuszczam funkcję...";
}

   print foo();
?>
```

Uruchomienie powyższego skryptu spowoduje wyświetlenie komunikatu „W funkcji”, następnie „1”, po czym skrypt zakończy działanie. Powodem, dla którego nigdy nie zostanie wyświetlony tekst „Opuszczam funkcję…”, jest fakt, że wiersz `return 1;` przekazuje wartość `1`, a następnie natychmiast kończy działanie funkcji. Drugie polecenie `print` w funkcji `foo()` nigdy nie zostanie osiągnięte.

Jeżeli chcesz przekazać z powrotem więcej niż tylko jedną wartość, musisz skorzystać z tablicy. To zagadnienie zostanie opisane w rozdziale 5.

Często spotykaną sytuacją jest zwracanie wartości instrukcji warunkowej, na przykład:

```
return $i > 10;
```

W powyższym poleceniu, jeśli zmienna `$i` będzie miała wartość większą od `10`, operator `>` zwróci `1`, co jest identyczne z zapisem `return 1;`. Natomiast w przypadku gdy zmienna `$i` jest mniejsza lub równa `10`, wówczas odpowiada to zapisowi `return 0;`.

Parametry

Projektowane przez Ciebie funkcje mogą korzystać z parametrów dzięki modyfikacji definicji funkcji pozwalającej na przyjęcie dowolnej liczby parametrów. Każdemu z parametrów możesz nadać nazwę, która będzie używana do odwołania się do parametru wewnątrz tej funkcji. Kiedy wywołasz później funkcję, PHP skopiuje wartości otrzymane dzięki tym parametrom, na przykład w taki sposób:

```
<?php
function mnozenie($liczba1, $liczba2) {
   $wynik = $liczba1 * $liczba2;
   return $wynik;
}

$moja_liczba = mnozenie(5, 10);
?>
```

Po uruchomieniu powyższego skryptu, zmienna `$moja_liczba` będzie miała ustawioną wartość `50`. Funkcja `mnozenie()` mogłaby zostać przepisana w taki sposób, aby zawierała po prostu jeden wiersz: `return $liczba1 * liczba2`. Przedstawiony przykład pokazuje, że możesz tworzyć swoje funkcje o dowolnej długości.

Przekazywanie przez referencję

Sprawy znacznie się komplikują, gdy dochodzimy do referencji, ponieważ musisz być w stanie zarówno przyjmować parametry przez referencję, jak i zwracać wartości przez referencję. Do tego celu wykorzystujemy operator referencji, którym jest znak `&`.

Oznaczenie parametru jako „przekazywanego przez referencję" następuje w definicji funkcji, a nie w wywołaniu funkcji. Dlatego też zapis:

```
function mnozenie(&$liczba1, &$liczba2) {
```

jest prawidłowy, podczas gdy zapis:

```
$moja_liczba = mnozenie(&5, &10);
```

jest błędny. To oznacza, że jeśli posiadasz wielokrotnie używaną funkcję w swoim projekcie, musisz przeprowadzić jedynie edycję definicji funkcji, aby pobierać zmienne przez referencję. Przekazywanie danych przez referencję jest często dobrym sposobem na skrócenie i uproszczenie czytania Twojego skryptu. Wybór rzadko jest powodowany przez czynniki związane z wydajnością. Rozpatrzmy poniższy kod:

```
function kwadrat1($liczba) {
   return $liczba * $liczba;
}

$wartosc = kwadrat1($wartosc);

function kwadrat2(&$liczba) {
```

```
    $liczba = $liczba * $liczba;
}

kwadrat2($wartosc);
```

Pierwszy przykład przekazuje kopię zmiennej `$wartosc`, mnoży ją, a następnie zwraca wynik, który jest kopiowany z powrotem do zmiennej `$wartosc`. W przykładzie drugim zmienna `$wartosc` zostaje przekazana przez referencję i zmodyfikowana bezpośrednio wewnątrz funkcji. Stąd zapis `kwadrat2($wartosc)` jest zupełnie wystarczający, zamiast kopiowania pierwszego przykładu.

Referencja jest *odniesieniem do zmiennej*. Jeżeli zdefiniowałeś funkcję, aby przyjmowała odwołanie do zmiennej, nie możesz przekazać do niej stałej. Oznacza to, że naszej definicji `kwadrat2()` nie możesz wywołać używając zapisu `kwadrat2(10);`. Liczba `10` nie jest zmienną, tak więc nie może zostać potraktowana referencyjnie.

Zwracanie referencji

W przeciwieństwie do przekazywania wartości przez referencję, co wymaga określenia referencyjnej natury parametru w definicji funkcji, w przypadku zwrócenia referencyjnego musisz wskazać to w definicji oraz w trakcie wywołania. Aby określić, że funkcja powinna zwracać referencyjnie, musisz umieścić operator referencji przed nazwą funkcji. Niezbędne jest również określenie, że wynik funkcji również powinien być referencyjny, w przeciwieństwie do jego kopiowania używanego w normalnym przypisaniu, które zostało opisane wyżej.

Poniżej znajduje się przykład takiej konstrukcji:

```
function &zwroc_rybe() {
    $ryba = "Wanda";
    return $ryba;
}

$ryba_ref =& zwroc_rybe();
```

Parametry domyślne

W trakcie projektowania swoich funkcji często pomocne jest przypisanie wartości domyślnych parametrom, które nie zostaną przekazane. PHP robi to w przypadku większości swoich funkcji; przeważnie oszczędza Ci to podawania parametrów, które są zwykle takie same.

W celu zdefiniowania swoich własnych wartości domyślnych dla parametrów funkcji, tuż za ustalaną zmienną, dodaj stałą wartość, która ma zostać przypisana, jeśli nie zostanie przekazany odpowiedni parametr. Na przykład:

```
function powitanie($imie = "Paweł") {
    return "Witaj $imie!\n";
}

powitanie();
powitanie("Paweł")
powitanie("Andrzej");
```

Uruchomienie powyższego skryptu spowoduje wyświetlenie:

```
Witaj Paweł!
Witaj Paweł!
Witaj Andrzej!
```

Zastanów się nad następującą funkcją:

```
function powitanie($imie, $nazwisko = "Kowalski") {}
```

Nie oznacza to, że zarówno zmienna `$imie`, jak i `$nazwisko` powinny przyjąć wartość `Kowalski`. Zamiast tego jedynie zmienna `$nazwisko` pobiera tę wartość — PHP traktuje te dwie zmienne jako funkcjonalnie niezależne od siebie, co oznacza, że możesz użyć poniższego kodu:

```
function powitanie($imie = "Jan", $nazwisko = "Kowalski") {
   return "Witaj, $imie $nazwisko!\n";
}
```

Możesz więc użyć tego kodu do powitania trzech zacnych osób o nazwiskach: Jan Kowalski, Tomasz Dawidowski i Tomasz Kowalski:

```
powitanie();
powitanie("Tomasz", "Dawidowski");
powitanie("Tomasz");
```

Jeżeli chciałbyś powitać osobę o nazwisku Jan Wilczak, byłoby idealnie, gdybyś pozwolił PHP na przekazanie za Ciebie pierwszego parametru. Jan jest bowiem wartością domyślną w tej funkcji, a Ty dostarczysz tylko nazwiska Wilczak. Gdy jednak spróbujesz uruchomić poniższy kod, to stwierdzisz, że on nie działa:

```
powitanie("Wilczak");
```

Zamiast otrzymać nazwisko Jan Wilczak, otrzymasz Wilczak Kowalski. Ponieważ PHP wypełnia parametry zaczynając od lewej strony, zostało przyjęte założenie, że przekazany przez Ciebie parametr był imieniem. Ta sama logika wskazuje, że nie możesz umieścić wartości domyślnej przed elementem nieposiadającym wartości domyślnej, na przykład w taki sposób:

```
function powitanie($imie = "Janek", $nazwisko) {}
```

Jeżeli ktokolwiek użyłby funkcji w taki sposób: `powitanie("Piotr")`, powstałoby pytanie, czy wówczas nastąpiłaby próba dostarczenia wartości dla zmiennej `$imie` zamiast użycia jej wartości domyślnej? A może użytkownik chciałby użyć wartości domyślnej dla imienia, natomiast `Piotr` to wartość dla zmiennej `$nazwisko`? Na szczęście PHP zgłosi błąd, jeśli nastąpiłaby próba użycia kodu w taki sposób!

Zliczanie parametrów zmiennej

Przedstawiona w rozdziale 7. funkcja `printf()` posiada możliwość pobrania dowolnej liczby parametrów — może pobrać po prostu jeden parametr lub pięć lub pięćdziesiąt lub pięćset. Liczba elementów możliwych do pobrania jest dowolna, o ile zostaną przekazane przez użytkownika. Jest to znane pod nazwą lista parametrów zmiennej i będzie automatycznie implementowane w funkcjach definiowanych przez użytkownika. Na przykład:

```
function pewna_funkcja($a, $b) {
   $j = 1;
}

pewna_funkcja(1, 2, 3, 4, 5, 6, 7, 8);
```

Przedstawiona powyżej funkcja `pewna_funkcja()` została tak zdefiniowana, aby pobierała jedynie dwa parametry `$a` i `$b`. Możemy ją jednak wywołać z ośmioma parametrami, a skrypt powinien się uruchomić bez problemów. Istnieje jeden aspekt, w którym PHP różni się znacznie od języka C: w C Twoje funkcje muszą zostać użyte ściśle w stosunku do deklaracji ich prototypów. W przedstawionym powyżej przykładzie wartość `1` zostanie umieszczona w zmiennej `$a`, a wartość `2` w zmiennej `$b`. Co się jednak stanie z pozostałymi parametrami?

Pomocne okażą się trzy funkcje: `func_num_args()`, `func_get_arg()` i `func_get_args()`, z których pierwsza i ostatnia nie pobierają parametrów. Aby pobrać liczbę parametrów, które zostały przekazane do Twojej funkcji, wywołaj funkcję `func_num_args()` i odczytaj jej wartość zwrotną. W celu pobrania wartości konkretnego parametru użyj funkcji `func_get_arg()` i przekaż jej numer parametru, którego wartość chcesz otrzymać z powrotem. Na końcu funkcja `func_get_args()` zwraca tablicę przekazanych parametrów. Poniżej znajduje się przykład użycia tych funkcji:

```
function pewna_funkcja($a, $b) {
   for ($i = 1; $i < func_num_args(); ++$i) {
      $parametr = func_get_arg($i);
      echo "Otrzymano parametr $parametr.\n";
   }
}

function pewna_inna_funkcja($a, $b) {
   $parametr = func_get_args();
   $parametr = join('', '', $parametr);
   echo "Otrzymano parametry: $parametr.\n";
}

pewna_funkcja(1, 2, 3, 4, 5, 6, 7, 8);
pewna_inna_funkcja(1, 2, 3, 4, 5, 6, 7, 8);
```

Używając funkcji `func_num_args()` możesz łatwo zaimplementować sprawdzanie błędów funkcji. Na przykład każdą swoją funkcję rozpocznij od upewnienia się, że funkcja `func_num_args()` zawiera oczekiwane przez Ciebie elementy, a jeśli tak nie jest, zakończ działanie funkcji. Kiedy jednak dodasz już `func_num_arg()`, powinieneś łatwo tworzyć swoje własne funkcje, które będą funkcjonowały z dowolną liczbą parametrów.

Zasięg zmiennej w funkcjach

Zmienne zdeklarowane na zewnątrz funkcji i klas są nazywane *globalnymi*, co oznacza, że są ogólnie dostępne w każdym miejscu skryptu. Ponieważ jednak funkcje są niezależnymi blokami, ich zmienne są niezależne i nie mają wpływu na zmienne w głównym skrypcie. W ten sam sposób zmienne z głównego skryptu nie są bezwarunkowo dostępne wewnątrz funkcji. Spójrz na poniższy przykład:

```
function foo() {
   $bar = "wombat[1]";
}

$bar = "baz";
foo();
print $bar;
```

Wykonanie powyższego skryptu rozpocznie się w wierszu `$bar = "baz"`, a następnie nastąpi wywołanie funkcji `foo()`. W tym momencie, jak możesz zobaczyć, funkcja `foo()` ustawi zmiennej `$bar` wartość `wombat`, po czym zwróci kontrolę do głównego skryptu, gdzie zmienna `$bar` zostanie wyświetlona. Funkcja `foo()` zostanie wywołana i nie wiedząc o istnieniu w zasięgu globalnym zmiennej `$bar`, utworzy w zasięgu lokalnym zmienną `$bar`. Kiedy działanie funkcji zostanie zakończone, wszystkie lokalne elementy zostaną odrzucone, pozostawiając nienaruszoną pierwotną zmienną `$bar`.

[1] Wombat — torbacz australijski, roślinożerne zwierzę zamieszkujące Australię i Tasmanię — *przyp. tłum.*

Ignorowanie zasięgu za pomocą tablicy GLOBALS

Superglobalna tablica `$GLOBAS` pozwala na dostęp do zmiennych globalnych, nawet z wewnątrz funkcji. Wszystkie zmienne zdeklarowane w zasięgu globalnym są umieszczone w tablicy `$GLOBALS`, do której masz dostęp z każdego miejsca skryptu. Poniżej znajduje się przykład użycia tablicy `$GLOBALS`:

```
function foo()
   $GLOBALS['bar'] = "wombat";
}

$bar = "baz";
foo();
print $bar;
```

Powyższy skrypt spowoduje wyświetlenie na ekranie słowa `wombat`, ponieważ funkcja `foo()` dosłownie zmieniła zmienną spoza jej zasięgu. Nawet gdy kontrola zostanie zwrócona do głównego skryptu, ten efekt wciąż pozostanie. Zmienne możesz odczytać w ten sam sposób:

```
$lokalny_bar = $GLOBALS['bar'];
```

Jest to jednak całkiem trudne do odczytania. PHP pozwala Ci na użycie specjalnego słowa kluczowego `GLOBAL`, aby zmienna mogła być dostępna lokalnie:

```
function moja_funkcja() {
   GLOBAL $foo, $bar, $baz;
   ++$baz;
}
```

Taki zapis pozwoli funkcji na odczytanie zmiennych globalnych `$foo`, `$bar` i `$baz`. Wiersz zawierający polecenie `++$baz` zwiększy wartość zmiennej `$baz` o `1`, co zostanie również odzwierciedlone w zasięgu globalnym.

Funkcje rekurencyjne

Czasami najłatwiejszym sposobem przedstawienia problemu jest spowodowanie, aby funkcja wywołała się samodzielnie — to technika zwana *rekurencyjnym wywołaniem funkcji*. Obliczanie silni jest tutaj najczęściej przytaczanym przykładem. Silnia liczby 6 wynosi 6 ∗ 5 ∗ 4 ∗ 3 ∗ 2 ∗ 1 lub 720, a zwykle jest przedstawiona jako „6!”. Tak więc silnia liczby 6 (6!) wynosi 720, „7!” wynosi „7 ∗ 6!”, musisz więc obliczyć jedynie wartość wyrażenia „6!”, a następnie wynik pomnożyć przez 7 w celu otrzymania „7!”.

Odpowiednie równanie może zostać zapisane w następujący sposób: „n! = n ∗ ((n–1)!)”. Oznacza to, że silnia dla danej liczby jest równa tej liczbie pomnożonej przez silnię liczby pomniejszonej o jeden — czysty przypadek funkcji rekurencyjnej. Potrzebujemy funkcji, która będzie przyjmowała liczbę całkowitą, i jeśli ta liczba nie jest zerem, wówczas nastąpi ponowne wywołanie funkcji. Tym razem jednak zostanie przekazana ta sama liczba, ale pomniejszona o jeden, a wynik będzie pomnożony przez nią samą. Poniżej znajduje się działający skrypt obliczający silnię:

```
function silnia($liczba) {
   if ($liczba == 0) return 1;
   return $liczba * silnia($liczba-1);
}

print silnia(6);
```

Wynikiem działania powyższego skryptu będzie wartość `720`, chociaż możesz łatwo edytować wywołanie funkcji `silnia()` i przekazać jej, na przykład, wartość `20` zamiast `6`. Wartość silni zwiększa się bardzo szybko („7!" wynosi 5040, „8!" wynosi 40320 itd.), tak więc ostatecznie osiągniesz ograniczenie przetwarzania. Nie chodzi tutaj o czas, ale jedynie o złożoność: PHP pozwala Ci tylko na pewien poziom rekurencyjności („18!" będzie maksimum, które będziesz mógł osiągnąć w trakcie obliczeń przy użyciu tego kodu).

Jak możesz się przekonać, funkcje rekurencyjne ułatwiają pewne zadania programistyczne i nie dotyczy to wyłącznie matematyki. Pomyśl, jak łatwo jest napisać funkcję `pokaz_odpowiedzi()` na potrzeby forum, która będzie automatycznie pokazywała wszystkie odpowiedzi na podaną wiadomość, wszystkie odpowiedzi do tych odpowiedzi oraz wszystkie odpowiedzi do tych odpowiedzi do tych odpowiedzi itd.