PHP. Stwórz własną stronę internetową

Autor: Praca zbiorowa
Tłumaczenie: Rafał Jońca (rozdz. 22 – 26, 28 – 33), Marek Pałczyński (rozdz. 3, 7, 10 – 13, 15), Ewa Sławińska (wprowadzenie, rozdz. 1, 2, 4 – 6, 8, 9, 14, 16 – 21, 27)
ISBN: 83-7361-665-9
Tytuł oryginału: Creating Your Web Site with PHP
Format: B5, stron: 496
Przykłady na ftp: 55 kB

Książka „PHP. Stwórz własną stronę internetową” to podręcznik przedstawiający zasady tworzenia aplikacji internetowych z wykorzystaniem języka PHP i bazy danych MySQL. Według założeń autora ma on pomóc nawet najmniej doświadczonym programistom w zdobyciu kwalifikacji niezbędnych do rozpoczęcia kariery zawodowej w świecie tworzenia aplikacji internetowych. Każde poruszone w nim zagadnienie ilustrowane jest starannie dobranymi przykładami metod programowania, które pomogą nauczyć się tworzenia wysokiej jakości aplikacji. Niemal we wszystkich rozdziałach książki prezentowane są najbardziej praktyczne i najczęściej wykorzystywane sztuczki programistyczne (mające związek nie tylko z językiem PHP).

- Podstawowe informacje o funkcjonowaniu internetu
- Zasady działania skryptów uruchamianych po stronie serwera
- Instalacja i konfiguracja serwera Apache, platformy PHP i bazy danych MySQL
- Szczegółowy opis języka PHP
- Korzystanie z bazy danych MySQL
- Tworzenie aplikacji modułowych
- Programowanie obiektowe w języku PHP

Zawarte w tej książce wiadomości pomogą każdemu programiście stworzyć doskonałą aplikację internetową.

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

Spis treści

Rozdział 9.

Konstrukcje składniowe języka PHP

W niniejszym rozdziale omówimy konstrukcje składniowe języka PHP. O niektórych z nich, na przykład o wyrażeniu `if`, była już okazja wspomnieć wcześniej. Tym razem jednak zapoznamy się z pełnym wachlarzem konstrukcji. Nie jest ich zbyt wiele, co na pewno należy zaliczyć do plusów języka PHP. Jak pokazuje praktyka, im bardziej zwięzła składnia języka programowania, tym łatwiej się nim posługiwać. PHP jest doskonałym przykładem prawdziwości tej teorii.

Struktura kontrolna if-else

Zaczynamy od omówienia najprostszej konstrukcji, mianowicie wyrażenia warunkowego. Jego składnia wygląda tak:

```
if(wyrażenie_logiczne)
  wyrażenie_1;
else
  wyrażenie_2;
```

Konstrukcja ta funkcjonuje tak, że gdy spełniony jest warunek `wyrazenie_logiczne`, to następuje wykonanie wyrażenia `wyrazenie_1`. Gdy wspomniany warunek nie zostanie spełniony, wykonywane jest wyrażenie `wyrazenie_2`. Podobnie jak ma to miejsce w innych językach programowania, konstrukcję `else` można opuścić. Wówczas w przypadku niespełnienia warunku nie będzie podejmowana żadna akcja.

Oto przykład:

```
  if($a>=1&&$b<=10) echo "Wszystko w porządku";
else echo "Nieprawidłowa wartość!";
```

Gdy zachodzi potrzeba zagnieżdżenia kilku wyrażeń w obrębie jednego, używa się do tego celu nawiasów klamrowych, jak w poniższym przykładzie:

```
if($a>$b) { print "a jest większe od b"; $c=$b; }
elseif($a==$b) { print "a jest równe b"; $c=$a: }
else {print "a jest mniejsze od b"; $c=$b; }
```

Uwaga — w powyższym skrypcie nie błędu! Zamiast `else if` można równie dobrze pisać `elseif` (czyli bez spacji pomiędzy tymi dwoma poleceniami), ale prawdopodobnie większość programistów uzna pierwszy sposób zapisu za bardziej czytelny.

Konstrukcja `if-else` ma także składnię alternatywną:

```
if(wyrażenie_logiczne):
  wyrażenia;
elseif(inne_wyrażenie_logiczne):
  inne_wyrażenia;
else:
  wyrażenia_alternatywne;
endif
```

Zwróćmy uwagę na dwukropek (`:`)! Jeśli zapomnimy umieścić go w skrypcie, wygenerowany zostanie komunikat o błędzie. Jak poprzednio, bloki `elseif` oraz `else` można pominąć.

Posługiwanie się składnią alternatywną

W poprzednich rozdziałach wielokrotnie rozpatrywaliśmy przykłady umieszczania kodu HTML w skryptach. W tym celu należy po prostu zamknąć skrypt znakiem ?>, wstawić pożądany fragment kodu HTML, po czym kontynuować skrypt po uprzednim zasygnalizowaniu jego początku znakiem <?.

Co prawda taki sposób zapisu nie wygląda zbyt elegancko, ale przy odrobinie wysiłku można to zmienić. Wystarczy sięgnąć po składnię alternatywną oraz inne konstrukcje języka.

Często zachodzi potrzeba umieszczenia kodu programu w obrębie dokumentu HTML, rzadziej dzieje się odwrotnie. Tak jest wygodniej, na przykład, dla projektanta grafiki strony, który jest odpowiedzialny za wygląd końcowego dzieła, a nie za pisanie skryptów. Dlatego właśnie powinno się dążyć do oddzielania kodu HTML od skryptu. Można to osiągnąć przez zapis kodu w odrębnym pliku, który będzie w odpowiednim momencie dołączany do skryptu w rezultacie użycia wyrażenia `include` (opis w dalszej części rozdziału). Do zagadnienia tego jeszcze powrócimy.

Poniżej przedstawiamy alternatywny sposób zapisu skryptu powitalnego zaprezentowanego w poprzednim rozdziale. Korzystamy z konstrukcji `if-else` (listing 9.1).

Listing 9.1. *Konstrukcja if-else. Alternatywny sposób zapisu*

```
  <?if(@$go) :?>
Cześć, <?=$name?>!
  <?else:?>
<form action=<?=$REQUEST_URI?> method=post>
Twoje imię: <input type=text name=name><br>
<input type=submit name=go value="Prześlij!">
  <?endif?>
```

Czytelnicy zapewne zgodzą się z tym, że skrypt w takiej postaci jest zrozumiały nawet dla osoby, która nie zna PHP, ale, na przykład, orientuje się w kodzie HTML.

Przedwarunkowa pętla while

Ta konstrukcja to kolejny spadek po języku C. Jej zadanie polega na wielokrotnym wykonywaniu wyrażeń zagnieżdżonych w pętli tak długo, jak długo spełniony będzie warunek wyrażony w sprawdzonym uprzednio wyrażeniu logicznym nagłówka pętli. W tym czasie konstrukcja wykonuje kolejne iteracje, ale natychmiast przerywa pracę, gdy tylko wyrażenie logiczne zwróci wartość `false`. Pętla ma następującą składnię:

```
while(wyrażenie_logiczne)
  wyrażenie;
```

Znaczenie powyższej definicji jest oczywiste; nazwa `wyrażenie_logiczne` mówi sama za siebie, natomiast `wyrażenie` to pojedyncze lub złożone wyrażenie wchodzące w skład pętli (oczywiście w celu niedopuszczenia do powstania pętli nieskończonej konieczne są pewne manipulacje mające na celu zmianę wartości wyrażenia — można w tym celu posłużyć się licznikiem). Jeśli wartością zwracaną przez wyrażenie będzie od początku `false`, to pętla nigdy nie zostanie wykonana. Oto przykład:

```
$i=1; $p=1;
while($i<32) {
  echo $p," ";
  $p=$p*2; // można by napisać $p*=2
  $i=$i+1; // można by napisać $i+=1 lub nawet $i++
}
```

Skrypt wyświetla liczby stanowiące wynik działania podnoszącego 2 do potęgi z zakresu od 1 do 31. Gdy zmiennej `$i` przypiszemy wartość `32`, żadna operacja nie zostanie wykonana.

Podobnie jak wyrażenie `if`, także pętla `while` ma składnię alternatywną, która ułatwia łączenie skryptów z kodem HTML:

```
while(wyrażenie_logiczne):
  wyrażenia;
endwhile;
```

Powarunkowa pętla do-while

W przeciwieństwie do pętli `while`, pętla `do-while` sprawdza wyrażenie logiczne *po* wykonaniu każdej iteracji, a nie przed. Z tego wniosek, że pętla ta musi być wykonana przynajmniej raz. Składnia wyrażenia jest następująca:

```
do {
  wyrażenia;
} while(wyrażenie_logiczne);
```

Po każdej iteracji sprawdzana jest prawdziwość wyrażenia logicznego — jeśli warunek jest spełniony, pętla wykonywana jest ponownie. W przeciwnym razie następuje przerwanie procesu.

Tego rodzaju pętla nie ma składni alternatywnej, co wynika najprawdopodobniej stąd, że ma ona nieczęste zastosowanie w skryptach.

Pętla uniwersalna for

Określenie „uniwersalna" nie wzięło się z powietrza — pętla `for` w PHP pozwala programistom tworzyć konstrukcje znacznie mniej trywialne w porównaniu z jej odpowiednikami z języków Pascal oraz C (tam pętla `for` służy jedynie do przechodzenia przez kolejne wartości zliczane przez licznik). Składnia pętli wygląda następująco:

```
for(wyrażenia_inicjujące; warunek_pętli; wyrażenia_po_iteracji)
  ciało_pętli;
```

Sprawdźmy teraz, jak to działa. W momencie uruchomienia pętli wykonywane są wyrażenia określone jako `wyrażenia_inicjujące` (w kolejności od lewej do prawej). Oddziela się je przecinkami, jak w poniższym przykładzie:

```
for($i=0,$j=10,$k="Test!"; .....)
```

Dalej rozpoczyna się iteracja. Najpierw sprawdzany jest warunek pętli (jak w konstrukcji `while`) i, w przypadku jego spełnienia (wartość `true` (prawda)), następuje wykonanie iteracji. Jeśli warunek nie zostanie spełniony, interpreter zakończy pętlę. Przykład:

```
// dodaje kolejną kropkę
for($i=0,$j=0,$k="Test"; $i<10; .....) $k.=".";
```

Załóżmy, że wykonana została jedna iteracja. W takim razie powinno dojść do wykonania wyrażeń po iteracji, których format jest identyczny jak wyrażeń inicjujących. Przykład:

```
for($i=0,$j=0,$k="Punkty"; $i<100; $j++,$i+=$j) $k=$k.".";
```

W tym miejscu należy wspomnieć, że powyższy przykład (jak również dowolny inny) można zaimplementować także z wykorzystaniem pętli `while`. Taka konstrukcja nie będzie jednak wyglądać równie dobrze jak pętla `for`. Oto przykład:

```
$i=0;$j=0;$k="Punkty";
while($i<100) {
  $k.=".";
  $j++;$i+=$j;
}
```

Cóż, to już chyba wszystko… Nie, chwileczkę! Spróbuj jeszcze zgadnąć, ile kropek zostanie dodanych do zmiennej `$k` po zakończeniu wykonywania pętli.

Pętla `for` — podobnie jak większość pozostałych konstrukcji — ma składnię alternatywną:

```
for(wyrażenia_inicjujące; warunek_pętli; wyrażenia_po_iteracji):
  wyrażenia;
endfor;
```

Wyrażenia break oraz continue

Kontynuujemy dyskusję na temat pętli. Czasem zachodzi potrzeba przerwania bieżącej iteracji (na przykład gdy zostanie spełniony określony warunek). W tym celu zaprojektowano wyrażenie `break`, które natychmiast kończy pracę pętli. Wyrażenie to może przyjmować parametr opcjonalny, służący do definiowania liczby zagnieżdżonych pętli, które należałoby przerwać. Domyślną wartością tego parametru jest `1` i wskazuje ona na pętlę bieżącą. Od czasu do czasu używa się jednak i innych wartości:

```
for($i=0; $i<10; $i++) {
  for($j=0; $j<10; $j++) {
    if($A[$i]==$A[$j]) break(2);
  }
}
if ($i<10) echo "W tablicy \$A znaleziono pasujące elementy!";
```

W powyższym przykładzie wyrażenie `break` przerywa zarówno drugą, jak i pierwszą pętlę, ponieważ jego parametr ma wartość `2`.

Opisywaną tutaj funkcję wprowadzono po raz pierwszy w czwartej wersji PHP. Autor niniejszej książki przyznaje, że nigdy dotąd nie pracował z językiem o tak przejrzystej składni — podziękowania dla twórców PHP.

Wyrażenie `break` przydaje się zwłaszcza wtedy, gdy trzeba coś wyszukać. W momencie spełnienia warunku wyszukiwania przez którąkolwiek z iteracji następuje przerwanie wykonywania pętli. Oto przykład pętli wyszukującej pierwszy zerowy element w tablicy `$A`:

```
for($i=0; $i<count($A); $i++)
  if($A[$i]==0) break;
if ($i<count($A)) echo "Znaleziono element zerowy: i=$i";
```

Nowością jest tutaj standardowa funkcja `count()`; zwraca ona liczbę elementów w tablicy.

Kolejne wyrażenie, `continue`, ma pewną cechę wspólną z wyrażeniem `break` — może być używane tylko „w kombinacji" z pętlą. Jego zadanie polega na natychmiastowym zakończeniu bieżącej iteracji i wykonaniu kolejnej (ale — co oczywiste — tylko w przypadku

spełnienia warunku postawionego w pętli przedwarunkowej). Kolejne podobieństwo do wyrażenia `break` polega na możliwości wskazania poziomu zagnieżdżenia pętli, do którego wykonywane będzie wyrażenie `continue`.

Jedną z najważniejszych zalet wyrażenia `continue` jest możliwość znacznego zredukowania liczby klamer, co wpływa na poprawę czytelności skryptu. Przydaje się to w pętlach filtrujących, gdzie wybiera się wyłącznie obiekty ściśle odpowiadające określonym warunkom. Oto przykład pętli, która przypisuje zera do elementów tablicy spełniających specyficzne warunki:

```
for($i=0; $i<count($A); $i++) {
  if(!warunek1($A[$i])) continue;
  ...
  if(!warunekN($A[$i])) continue;
  $A[$i]=0;
}
```

Poprawne korzystanie z wyrażeń `break` oraz `continue` może pozytywnie wpłynąć na czytelność skryptu, zmniejszając przy okazji ogólną liczbę bloków `else`. Oczywiście przedstawione tutaj przykłady ich użycia nie są najwyższych lotów, ale pozwalają się zorientować, że prędzej czy później każdy programista dojdzie do wniosku, że nie potrafi się bez nich obejść.

Niezwykłe zastosowania wyrażeń do-while oraz break

Stosowanie wyrażenia `break` daje ciekawy efekt uboczny, przydatny zwłaszcza wówczas, gdy trzeba ominąć w skrypcie „niepotrzebne" elementy (nawiasem mówiąc, efekt ten da się również wykorzystać w języku C). Taka potrzeba omijania zdarza się zadziwiająco często. Rozważmy następujący przykład (listing 9.2).

Listing 9.2. *Przykładowy skrypt do przetwarzania formularzy*

```
...
$WasError=0; // Wskaźnik błędu: wartość inna niż zero oznacza błąd
// Gdy zostanie kliknięty przycisk potwierdzenia (o nazwie $doSubmit)...
if(@$doSubmit) do {
  // Sprawdzanie danych wejściowych
  if(nieprawidłowa_nazwa_użytkownika) { $WasError=1; break; }
  ...
  if(nieprawidłowe_dane) { $WasError=1; break; }
  ...
  // Dane są prawidłowe, przetwórz je
  wykonaj działania;
  wyświetl rezultaty;
  zakończ skrypt;
} while(0);
...
  // Wyświetl formularz, z którego użytkownik skorzysta w celu uruchomienia skryptu.
  // Możliwe, że należy wyświetlić komunikat o błędzie, jeśli $WasError!=0
```

Jest to typowy sposób pisania skryptów dialogowych. Skrypt uruchamiany jest bez żadnych parametrów i użytkownik widzi formularz pytający o jego imię, hasło i inne tego typu informacje. Po kliknięciu przycisku następuje wykonanie operacji zaprogramowanych w skrypcie. Polegają one na sprawdzeniu, czy kliknięto przycisk `doSubmit`, a także czy podano nazwę i hasło. Jeśli dane te są nieprawidłowe, formularz wyświetlany jest ponownie (z komunikatem o błędzie podświetlonym na czerwono). Gdy zaś wszystko jest w porządku, skrypt kończy swoje działanie i wyświetla rezultaty.

Jak widać, przedstawioną tutaj logikę dość łatwo zaadaptować do konkretnej sytuacji, o ile tylko znamy metodę przerywania wykonywania bloku sprawdzającego i ponownego wyświetlania formularza. Osiągamy to przez użycie konstrukcji:

```
if(warunek) do {...} while(0);
```

Oczywiście pętla `do-while` wykonywana jest tylko raz, jako że jej warunek jest zawsze fałszywy. Takiej „zwyrodniałej" pętli można jednak użyć w sytuacji, gdy zajdzie potrzeba zakończenia jej wyrażeniem `break`.

Niektórzy Czytelnicy mogą w tym momencie stwierdzić, że w podobnych przypadkach bardziej odpowiednie byłoby użycie funkcji i wyrażenia `return`, lecz to jest właśnie przykład sytuacji, w której korzystanie z funkcji nie jest zbyt wygodne. Chodzi o to, że aby pozyskać zmienną globalną (w rodzaju elementu formularza) z funkcji, trzeba wykonać dodatkowe operacje. Jest to oczywiście wadą PHP i wkrótce przedyskutujemy ją szerzej.

Pętla foreach

Ten rodzaj pętli zaprojektowano specjalnie z myślą o przeglądaniu poszczególnych elementów tablicy. Ma ona następującą składnię:

```
foreach(tablica as $klucz=>$wartość)
  wyrażenia;
```

W powyższym przypadku wyrażenia są wykonywane kolejno dla każdego elementu tablicy, a bieżąca para `klucz=>wartość` przypisywana jest do zmiennych `$klucz` oraz `$wartość`. Rozważmy teraz inny przykład (listing 9.3), w którym do wyświetlenia wszystkich zmiennych globalnych posługujemy się konstrukcją `foreach`.

Listing 9.3. *Wyświetlanie zmiennych globalnych*

```
<?
foreach($GLOBALS as $k=>$v)
echo "<b>$k</b> => <tt>$v</tt><br>\n";
?>
```

Pętla `foreach` ma składnię alternatywną, przydatną zwłaszcza wtedy, gdy nie potrzebujemy wartości klucza bieżącego elementu. Ten alternatywny format zapisu przedstawia się następująco:

```
foreach(tablica as $wartość)
  wyrażenia;
```

W tym przypadku dostępna jest wyłącznie *wartość* elementu, natomiast klucz — nie. Jest to użyteczne rozwiązanie, na przykład, podczas pracy z tablicami list.

Pętla `foreach` nie korzysta z prawdziwej tablicy, lecz z jej kopii. Oznacza to, że wszelkie zmiany wprowadzone w tablicy nie będą odzwierciedlane w pętli. Można jednak użyć rezultatu wykonania funkcji jako tablicy. W takim wypadku funkcję wywołuje się tylko raz, przed rozpoczęciem pętli, a wszelkie następujące później manipulacje są już wykonywane na kopii zwróconej wartości.

W następnym rozdziale przyjrzymy się nieco uważniej zagadnieniom dotyczącym tablic asocjacyjnych.

Struktura kontrolna switch-case

W wielu wypadkach wygodniej jest posłużyć się specjalną konstrukcją `switch-case` w miejsce rozbudowanych wyrażeń `if-else`:

```
switch(wyrażenie) {
  case wartość1: wyrażenia1; [break;]
  case wartość2: wyrażenia2; [break;]
  ...
  case wartośćN: wyrażeniaN; [break;]
  [default: domyślne_wyrażenia; [break;]]
}
```

Konstrukcja `switch-case` działa w sposób następujący: najpierw szacowana jest wartość wyrażenia (załóżmy, że jest to wartość `V`), a następnie wyszukiwana jest linia rozpoczynająca się od `case V:`. Po jej odnalezieniu wykonywane są kolejne wyrażenia (w tym czasie wszystkie pojawiające się dalej wyrażenia typu `case` są ignorowane, w przeciwieństwie do pozostałych fragmentów kodu). Gdy pożądana linia nie zostanie odnaleziona, wykonywane są wyrażenia pojawiające się po wyrażeniu `default` (o ile jakieś istnieją).

Zwróćmy uwagę na obecność wyrażeń `break` (zapisano je w nawiasach, co oznacza, że są opcjonalne). Wstawia się je po każdej grupie wyrażeń (`wyrażenia1`, `wyrażenia2` itd.) z wyjątkiem ostatniej — dodanie wyrażenia `break` poniżej ostatniej linii jest możliwe, ale nie ma większego sensu. W przypadku zaistnienia warunku `V=wartość1` mogą być wykonane wszystkie wyrażenia, a nie tylko te z grupy pierwszej (`wyrażenia1`).

Składnia alternatywna dla konstrukcji `switch-case` prezentuje się następująco:

```
switch(wyrażenie):
  case wartość1: wyrażenia1; [break;]
  ...
  case wartośćN: wyrażeniaN; [break;]
  [default: domyślne_wyrażenia; [break;]]
endswitch;
```

Wyrażenie require

Wyrażenie `require` umożliwia dokonanie podziału tekstu skryptu na części, z których każda tworzy oddzielny plik. Składnia wyrażenia wygląda następująco:

```
require nazwa_pliku;
```

Rozpoczynając pracę (ale nie wykonanie programu!), interpreter zastępuje wyrażenie plikiem o nazwie `nazwa_pliku` (może to być plik zawierający skrypt PHP ograniczony znacznikami <? i ?>). Co w tym wszystkim najważniejsze, operacja zastępowania wyrażenia plikiem wykonywana jest *tylko jeden raz* (czyli zupełnie inaczej niż ma to miejsce w przypadku wyrażenia `include` opisywanego w dalszej części rozdziału) przed rozpoczęciem wykonywania programu. To bardzo wygodne rozwiązanie, szczególnie gdy zachodzi potrzeba dołączenia do skryptu różnych nagłówków i (lub) stopek HTML. Przykłady znajdziemy na listingach 9.4, 9.5 i 9.6.

Listing 9.4. *Plik naglowek.htm*

```
<html>
<head><title>Temat!</title></head>
<body bgcolor=yellow>
```

Listing 9.5. *Plik stopka.htm*

```
&copy;Moja firma, 1999.
</body></html>
```

Listing 9.6. *Plik skrypt.php*

```
<?
require "naglowek.htm";
... skrypt wyświetlający dokument
require "stopka.htm";
?>
```

Jest to o wiele lepsze rozwiązanie od zwykłego przetykania wyrażeń skryptu fragmentami kodu HTML. Umożliwia, na przykład, wygodną zmianę wyglądu strony osobom, które korzystają z usług naszego programu. Z drugiej strony, nie jest to programowanie w dobrym stylu — w końcu mamy tu do czynienia z aż trzema różnymi plikami! Tymczasem, jak wspomniano wcześniej, im mniej plików składa się na cały program, tym łatwiej nim operować (szczególnie z punktu widzenia osoby, która nie potrafi posługiwać się PHP, czyli na przykład projektanta graficznego).

Przy okazji lektury rozdziału 30. poznamy technikę rozdzielania kodu i szablonów, która będzie znakomitym lekarstwem na tego typu problemy.

Wyrażenie include

Wyrażenie `include` jest niemal identyczne z wyrażeniem `require`, z jednym wszakże wyjątkiem. Otóż dołączany plik staje się częścią skryptu nie przed jego wykonaniem, lecz właśnie w czasie wykonywania.

Jak to różnica? Już wyjaśniamy. Załóżmy, że mamy dziesięć plików tekstowych, nazwanych odpowiednio *plik0.php*, *plik1.php*, *plik2.php* i tak dalej, aż do *plik9.php*. Każdy z tych plików zawiera cyfrę z zakresu od 0 do 9. Uruchamiamy więc następujący program:

```
for($i=0; $i<10; $i++) {
  include "plik$i.php";
}
```

i w rezultacie jego wykonania otrzymujemy zestaw dziesięciu cyfr: `"0123456789"`. Widać z tego, że podczas wykonywania pętli każdy z plików tekstowych został jednokrotnie dołączony do skryptu (spróbujmy teraz zastąpić wyrażenie `include` wyrażeniem `require`, a następnie porównajmy rezultaty).

Być może Czytelnik zwrócił uwagę na to, że klamry wydają się tu zbędne. Ale to pozory. Gdy tylko je usuniemy, zgłoszony zostanie dziwny komunikat o błędzie (lub, co gorsza, program zacznie funkcjonować nieprawidłowo bez żadnej widocznej przyczyny). Dlaczego tak się dzieje? Ponieważ wyrażenie `include` nie jest bynajmniej wyrażeniem sensu stricte. Aby to zrozumieć, wyobraźmy sobie, że za każdym razem, gdy interpreter napotka wyrażenie `include`, po prostu dołącza do skryptu zawartość odpowiedniego pliku. Lecz co się stanie, jeśli ów plik będzie zawierać więcej niż jedno wyrażenie? Tylko pierwsze z nich zostanie wykonane podczas odtwarzania pętli, natomiast pozostałe zostaną uwzględnione dopiero *po* zakończeniu pętli. Reguła jest zatem taka: jeśli tylko używamy wyrażenia `include` w obrębie inne konstrukcji, to powinniśmy otoczyć je klamrami.

Istnieje możliwość, że w nowszych wersjach PHP sytuacja ulegnie poprawie, ale raczej trudno uznać to za pewnik.

Translacja i problemy z użyciem wyrażenia include

Jak już wiemy, skrypt PHP jest przed wykonaniem przekształcany na jego wewnętrzną reprezentację. Po zakończeniu tego procesu w pamięci komputera znajduje się już na wpół „upieczony” program, bez komentarzy, niektórych nazw zmiennych, formatowania itd. Potem następuje interpretacja, czyli wykonanie tej wewnętrznej reprezentacji skryptu. Wiemy jednak, że mogą istnieć takie fragmenty programu, których nie da się poddać translacji z wyprzedzeniem — jest ona wykonywana później, gdy interpreter napotka takie fragmenty podczas wykonywania skryptu.

Jednym z takich „trudnych" fragmentów kodu jest wyrażenie `include`. Gdy tylko interpreter napotka je w skrypcie, zatrzymuje jego wykonanie i wywołuje translator, którego zadaniem jest przetworzenie zawartości dołączanego pliku. Zmniejsza to szybkość działania programu, szczególnie gdy mamy do czynienia z naprawdę dużym plikiem. Wniosek z tego taki, że gdy pracujemy nad rozległym i bardzo złożonym skryptem, powinniśmy unikać wyrażenia `include`, zastępując je — gdzie to tylko możliwe — wyrażeniem `require`.

Powyższa rada zyskuje na znaczeniu, jako że mamy w perspektywie możliwość pojawienia się któregoś dnia kompilatora PHP. Cechowałby się on zdolnością przechowywania przetłumaczonego kodu w plikach wykonawczych (tak jak ma to miejsce w programach tworzonych w języku Perl). Wówczas użycie wyrażenia `include` powodowałoby zamieszanie, bo kompilator nie „wiedziałby", gdzie szukać żądanych plików i ostatecznie nie mogłyby one trafić do plików wykonawczych.

Które z wyrażeń jest zatem lepsze: `require` czy `include`? Gdy jesteśmy pewni, że dany plik powinien być dołączony tylko raz i w jednym konkretnym miejscu, skorzystajmy z wyrażenia `require`. W innym wypadku korzystniejsze może być użycie wyrażenia `include`.

Wyrażenia typu _once

Wyrażeń `include` i `require` używa się najczęściej w rozległych i złożonych skryptach. Z tego względu odnajdywanie wszystkich miejsc wstawienia tego samego pliku może sprawiać nie lada trudności (nie wspominając już o wynikających z tego błędach).

Żeby mieć lepsze wyobrażenie na temat problemu, z którym mamy do czynienia, rozważymy taką oto historię. Pewnego dnia programista Grzegorz napisał kilka użytecznych funkcji do obsługi plików programu Excel. Następnie podjął decyzję, że zintegruje je z biblioteką o nazwie *xllib.php* (listing 9.7).

Listing 9.7. *Biblioteka xllib.php*

```
<?
function LoadXlDocument($filename) {...}
function SaveXlDocument($filename,$doc) {...}
?>
```

Programista Antek postanowił stworzyć podobne funkcje dla programu Word, co zaowocowało powstaniem kolejnej biblioteki, o nazwie *wlib.php*. Ponieważ programy Excel i Word mają ze sobą wiele wspólnego, Antek postanowił skorzystać z biblioteki *xllib.php*, odwołując się do niej za pomocą wyrażenia `require` (listing 9.8).

Nie minęło wiele czasu i obydwie biblioteki zyskały sobie taką popularność, że zaczęli z nich korzystać inni programiści tworzący na potrzeby sieci. Przy tym nikt nie zawracał sobie głowy konstrukcją bibliotek; po prostu dołączali je do skryptów za pomocą wyrażenia `require`.

Listing 9.8. *Biblioteka wlib.php*

```
<?
require "xllib.php";
function LoadWDocument($filename) {...}
function SaveWDocument($filename,$doc) {...}
?>
```

Pewnego pięknego dnia, bliżej nikomu nieznany programista postanowił popracować zarówno z dokumentami Excela, jak i Worda. Nie namyślając się wiele, dołączył do skryptu obydwie biblioteki (listing 9.9).

Listing 9.9. *Dołączanie bibliotek xllib.php i wlib.php*

```
<?
require "wlib.php";
require "xllib.php";
$wd=LoadWDocument("dokument.doc");
$xd=LoadXlDocument("dokument.xls");
?>
```

Wyobraźmy sobie zdziwienie na twarzy tego programisty, gdy po uruchomieniu skryptu jego oczom ukazał się komunikat o błędzie. Komunikat ten głosił, że funkcja `LoadXlDocument()` w pliku *`xllib.php`* została zdefiniowana więcej niż raz!

W czym tkwi problem? Łatwo to ustalić, gdy prześledzi się sposób, w jaki translator PHP „tworzy" kod z listingu 9.9. Spójrzmy:

```
//require "wlib.php";
  //require "xllib.php";
    function LoadXlDocument($filename) {...}
    function SaveXlDocument($filename,$doc) {...}
  function LoadWDocument($filename) {...}
  function SaveWDocument($filename,$doc) {...}
//require "xllib.php";
  function LoadXLDocument($filename) {...}
  function SaveXLDocument($filename,$doc) {...}
$wd=LoadWDocument("dokument.doc");
$wd=LoadXlDocument("dokument.xls");
```

Jak widać, plik *`xllib.php`* został dołączony dwukrotnie: najpierw — pośrednio — przez bibliotekę *`wlib.php`*, a potem już bezpośrednio przez sam program. Dlatego właśnie translator zgłosił błąd, gdy napotkał linię oznaczoną w powyższym skrypcie pogrubioną czcionką.

Oczywiście autor nieszczęsnego skryptu mógłby wcześniej przeanalizować plik źródłowy biblioteki *`wlib.php`* i wówczas na pewno odkryłby, że nie powinien dopuścić do dwukrotnego dołączenia pliku *`xllib.php`*. Nie takiego jednak rozwiązania oczekujemy. Gdy dochodzi do pośredniego, zagnieżdżonego dołączania plików, jedynym sposobem na rozwikłanie problemu wydaje się modyfikacja bibliotek. Ale tego nie chcemy. Co więc powinniśmy zrobić?

Cóż, po tak długim wprowadzeniu (ale chyba nie za długim?), czas poznać opinię twórców PHP. Otóż oferują oni bardzo eleganckie rozwiązanie wszystkich naszych bolączek: wyrażenia `include_once` oraz `require_once`.

Wyrażenie `require_once` działa niemal identycznie jak wyrażenie `require` z jednym tylko, lecz zasadniczym, wyjątkiem. Gdy okaże się, że żądany plik został już wcześniej dołączony do skryptu, wyrażenie nie robi nic. Oczywiście takie rozwiązanie wymaga od interpretera PHP przechowywania pełnych nazw wszystkich dołączonych plików. I tak też się dzieje.

Podobnie rzecz się ma z wyrażeniem `include_once` z tą różnicą, że działa ono w fazie wykonywania programu, a nie w fazie jego translacji.

Jak już wspomniano, w PHP dostępna jest wewnętrzna tablica, w której przechowywane są pełne nazwy wszystkich dołączonych plików. Zawartość tej właśnie tablicy sprawdzana jest przez wyrażenia `include_once` oraz `require_once`. Co więcej, *dodawanie* wartości do tej tablicy jest także realizowane przez wyrażenia `require` i `include`. I właśnie dlatego po dołączeniu pliku do skryptu, na przykład przez wyrażenie `require`, żądanie ponownego dołączenia tego samego pliku przez wyrażenie `require_once` jest po prostu ignorowane.

Wyrażeń z przyrostkiem `_once` starajmy się używać wszędzie, gdzie to tylko możliwe, natomiast unikajmy jak ognia wyrażeń `require` oraz `include`. W ten sposób łatwiej nam będzie dzielić złożone skrypty na mniejsze i względnie niezależne fragmenty.