Struktury kontrolne - Podstawy PHP

Programy wykonujące szereg zawsze tych samych instrukcji nie pozwalają rozwinąć skrzydeł. Jeżeli aplikacja ma być w pełni interaktywna, musi umieć reagować na poczynania użytkownika lub sytuacje kryzysowe na podstawie zadanych warunków. Tu do akcji wkraczają instrukcje kontrolne. Są one podstawowym budulcem wielu algorytmów i ich poznanie jest niezbędne. Za to my zyskamy pełną kontrolę nad wszystkim, mogąc efektywnie reagować na wszystkie zdarzenia.

Jedną z instrukcji kontrolnych - **if** - widziałeś już w poprzednim rozdziale. Zauważyłeś też pewnie, iż nie była ona zakończona średnikiem, a zamiast tego pojawiały się przy niej nawiasy klamrowe tworzące tzw. **blok kodu**. Jest to ciąg komend złożony z wyrażeń i innych struktur kontrolnych, czytelnie oznakowany. W połączeniu z odpowiednią instrukcją, PHP może decydować, czy wykonać go, czy nie, czy też powtórzyć raz jeszcze. Klamry pełnią tutaj rolę drogowskazu pokazującego, co się danej instrukcji tyczy.

PHP posiada siedem struktur kontrolnych, lecz my poznamy sześć z nich. Siódma jest tak rzadko stosowana, że większość zaawansowanych programistów nie umie z niej korzystać, a ponadto wymaga ona od nas pewnej dodatkowej wiedzy. Oto lista wszystkich instrukcji:

Instrukcja if

Z tą instrukcją zetknęliśmy się już przy okazji omawiania formularzy. Przypomnimy jeszcze raz ten przykład:

```
<?php

if(count($_GET) == 2)
{
echo 'Witaj, '.$_GET['imie'].' '.$_GET['nazwisko'].'!';
}
else
{
echo 'Nieprawidłowa liczba parametrów!';
}

?>
```

Instrukcja if pozwala na wykonanie części kodu tylko wtedy, kiedy spełniony jest określony warunek i opcjonalne dodawanie alternatywnych instrukcji w razie jego fałszywości. Z *if*-a korzysta się niezwykle często, ponieważ niemal zawsze musimy sprawdzać, czy dane informacje są prawidłowe, czy funkcja poprawnie połączyła się z plikiem itd. Poniżej napiszemy skrypt, który będzie potrafił rozwiązywać równanie kwadratowe $y = ax^2 + bx + c$.
Przypomnijmy, że ilość jego rozwiązań rzeczywistych zależy od wartości tzw. współczynnika Δ ($\Delta = b^2 - 4ac$). Jeżeli obliczony wynik (Δ) jest dodatni, istnieją dwa rozwiązania. Jeżeli ujemny - nie ma żadnych. Dla zera równanie daje jedno rozwiązanie (podwójne).

```
<?php
// 1
```

```
  if(!isset($_GET['a']))
{
$_GET['a'] = ;
}
if(!isset($_GET['b']))
{
$_GET['b'] = ;
}
if(!isset($_GET['c']))
{
$_GET['c'] = ;
}

// 2
  if($_GET['a'] == )
{
die('Nieprawidłowy parametr A!');
}

// 3
  $delta = pow($_GET['b'], 2) - 4 * $_GET['a'] * $_GET['c'];

// 4
  if($delta > )
{
// 5
    echo 'Delta dodatnia. Dwa rozwiązania:<ul>';
echo '<li>'.round((-$_GET['b']-sqrt($delta))/(2*$_GET['a']), 5).'</li>';
echo '<li>'.round((-$_GET['b']+sqrt($delta))/(2*$_GET['a']), 5).'</li>';
echo '</ul>';
}
elseif($delta < )
{
// 6
    echo 'Delta ujemna. Brak rozwiązań!';
}
else
{
// 7
    echo 'Delta = 0. Jedno rozwiązanie: '.round((-$_GET['b'])/(2*$_GET['a']));
}
?>
```

Jest to pierwszy tak długi kod zawarty w tym podręczniku, dlatego omówimy go sobie w punktach. Numery odpowiednich fragmentów zaznaczone są w kodzie komentarzami jednolinijkowymi.

1. Na początek sprawdzamy, czy użytkownik podał wszystkie parametry. Funkcja *isset()* zwraca wartość **TRUE**, jeżeli zmienna istnieje, a operator negacji (**!**) sugeruje, że kod w nawiasie chcemy wykonać wtedy, gdy tej zmiennej nie ma. Musimy wtedy podstawić za nią neutralną wartość 0, ponieważ inaczej skrypt będzie nam zgłaszać powiadomienia.
2. Tutaj zaczynamy właściwy algorytm. Najpierw sprawdzamy, czy rzeczywiście mamy do czynienia z równaniem kwadratowym. Parametr *a* musi być różny od zera. W razie problemów instrukcją **die()** zatrzymujemy skrypt w tym miejscu.
3. Liczymy współczynnik Δ. Funkcja *pow(liczba, potega)* podnosi podaną liczbę do odpowiedniej potęgi i działa szybciej, niż ręczne mnożenie wartości.
4. Pierwszy wariant - kiedy Δ jest dodatnia...

5. Oblicz każde z dwóch rozwiązań równania odpowiednim wzorem. Funkcja *round(liczba, miejsca)* zaokrągla nam wynik do określonej ilości miejsc po przecinku, natomiast *sqrt(liczba)* zwraca pierwiastek z podanej liczby. Zwróć uwagę na użycie nawiasów do zasugerowania właściwej kolejności działań.
6. Gdy Δ jest ujemna, równanie nie ma rozwiązania.
7. Ostatni z wariantów jest oczywisty, dlatego nie piszemy już warunku. To przecież ostatnia z możliwości. Równanie ma tylko jedno rozwiązanie i także je wyliczamy.

Jest to nasz pierwszy prawdziwie dynamiczny skrypt, który potrafi reagować inaczej w zależności od sytuacji. Poznaliśmy tutaj nie tylko kilka nowych funkcji, ale także sporo operatorów i zasadę działania instrukcji *if*. Jej formalna składnia jest następująca:

```
<?php
if(wyrazenie)
{
// blok kodu
  }
elseif(wyrazenie)
{
// blok kodu
  }
else
{
// blok kodu
  }
?>
```

Obowiązkowe jest podawanie pierwszego z członów zaczynającego się od *if*. Dwa pozostałe są opcjonalne, przy czym ilość *elseif* może być dowolna. Kolejność podawania kolejnych typów członów ukazana jest na przykładzie.

- *if* - wykonuje się, gdy spełniony został podany warunek
- *elseif* - jeżeli nie został spełniony poprzedni warunek, PHP testuje aktualny i jeżeli jest prawdziwy, wykonuje ten kawałek kodu.
- *else* - wykonywane, jeżeli żaden z powyższych warunków nie został spełniony.

Jeżeli blok kodu zawiera tylko jedną instrukcję, PHP dopuszcza możliwość opuszczenia nawiasów klamrowych, np.

```
<?php
if($zmienna == 6)
echo 'Wartość zmiennej wynosi 6';
?>
```

W tym podręczniku jednak nie będziemy jej stosować z powodu pogorszenia czytelności kodu, niemniej warto wiedzieć, iż składnia ta jest w pełni poprawna, ponieważ część programistów stosuje ją w praktyce.

Wyrażenie warunkowe powinno przyjmować wartości logiczne **TRUE** lub **FALSE**. Oto kilka przydatnych operatorów:

Operator	Nazwa	Składnia	Opis
==	Równość	wyrażenie == wyrażenie	Zwraca prawdę, jeżeli oba wyrażenia mają identyczną wartość.
===	Równość	wyrażenie === wyrażenie	Zwraca prawdę, jeżeli oba wyrażenia mają identyczną wartość **oraz typ**.
!=	Nierówność	wyrażenie != wyrażenie	Zwraca prawdę, jeżeli oba wyrażenia mają różne wartości.
!==	Nierówność	wyrażenie !== wyrażenie	Zwraca prawdę, jeżeli oba wyrażenia mają różne wartości i/lub typ.
<	Mniejsze niż	wyrażenie < wyrażenie	Zwraca prawdę, jeżeli lewe wyrażenie ma mniejszą wartość od prawego.

>	Większe niż	wyrażenie > wyrażenie	Zwraca prawdę, jeżeli lewe wyrażenie ma większą wartość od prawego.
<=	Mniejsze lub równe	wyrażenie <= wyrażenie	Zwraca prawdę, jeżeli lewe wyrażenie ma mniejszą lub równą wartość prawemu.
>=	Większe lub równe	wyrażenie >= wyrażenie	Zwraca prawdę, jeżeli lewe wyrażenie ma większą lub równą wartość prawemu.
!	Negacja (nie)	!wyrażenie	Zwraca prawdę, jeżeli wyrażenie jest fałszywe i fałsz, jeśli prawdziwe.
&&	Koniunkcja logiczna (i)	wyrażenie && wyrażenie	Zwraca prawdę, jeżeli oba wyrażenia są prawdziwe.
||	Alternatywa logiczna (lub)	wyrażenie || wyrażenie	Zwraca prawdę, jeżeli przynajmniej jedno z wyrażeń jest prawdziwe.

A teraz kilka przykładów...

```
<?php
$liczba1 = $_POST['liczba1']; //Zakładamy, że do skryptu wysłano formularz z polami liczba1
$liczba2 = $_POST['liczba2']; // i liczba2. Pobieramy je...

if($liczba1 == 1 && $liczba2 == 2)
{
die('Liczba 1 wynosi 1, a liczba 2 wynosi 2');
}
?>
```

Powyższy skrypt sprawdza, czy **zmienna "liczba1"** wynosi *1* **i zmienna "liczba 2"** wynosi *2*.

```
<?php
$liczba1=$_POST['liczba1']; //Zakładamy, że do skryptu wysłano formularz z polami liczba1
$liczba2=$_POST['liczba2']; // i liczba2. Pobieramy je...

if($liczba1==1 || $liczba2==1)
{
die('Liczba 1, lub liczba 2 wynosi 1');
}
?>
```

Natomiast ten skrypt sprawdza, czy **zmienna "liczba1"**, **lub zmienna "liczba2"** wynosi *1*.

Wszystkie operatory podane wcześniej w tabelce przydają się przy konstruowaniu warunków. Pewnego wyjaśnienia domagają się **==** oraz **===**. Popatrz sobie na taki przykład:

```
<?php
if(FALSE == )
{
echo 'Prawda!';
}
?>
```

PHP automatycznie sprowadzi tu sobie obie wartości do identycznego typu i wtedy dopiero je porówna. Dlatego skrypt wyświetli napis "Prawda!". Zamień teraz ten operator na **===**. Po odświeżeniu zobaczymy, że teraz nic się nie pokazało. To dlatego, że zażądaliśmy, aby i typy obu wyrażeń były identyczne, podczas gdy nie są. Operator ten przydaje się przy niektórych funkcjach zwracających różne typy wartości w zależności od powodzenia operacji.

Uwaga!
Operatory **=** i **==** są w PHP bardzo podobne, dlatego czasem przy ich wpisywaniu zdarzają się pomyłki. Jeżeli twoja instrukcja warunkowa zachowuje się tak, jakby jej warunek był zawsze prawdziwy, upewnij się, że wstawiłeś tam właściwy operator!

Oprócz tego w warunkach przyda się nam kilka funkcji:

- *isset($zmienna)* - zwraca prawdę, jeżeli zmienna istnieje.
- *empty($zmienna)* - zwraca prawdę, jeżeli zmienna ma wartość inną od **NULL**.
- *is_null($zmienna)* - zwraca prawdę, jeżeli zmienna ma wartość **NULL**.
- *is_string($zmienna)* - zwraca prawdę, jeżeli zmienna jest ciągiem tekstowym.
- *is_integer($zmienna)* - zwraca prawdę, jeżeli zmienna jest liczbą całkowitą.
- *is_float($zmienna)* - zwraca prawdę, jeżeli zmienna jest liczbą zmiennoprzecinkową.
- *is_numeric($zmienna)* - zwraca prawdę, jeżeli zmienna jest liczbą.
- *is_bool($zmienna)* - zwraca prawdę, jeżeli zmienna ma wartość logiczną.
- *is_array($zmienna)* - zwraca prawdę, jeżeli zmienna jest tablicą.

Instrukcja switch

Instrukcja switch zwana jest także instrukcją wyboru. Jej działanie jest podobne do szczególnego przypadku poznanej ostatnio instrukcji warunkowej. Pokażemy to na przykładzie.

Nietrudno znaleźć witryny internetowe, które w jednym pliku grupują kilka różnych zadań wykonywanych w zależności od parametru, np. "index.php?act=dodaj", "index.php?act=usun" itd. Możemy to zaprogramować za pomocą dużego ifa:

```
<?php

if($_GET['act'] == 'dodaj')
{
echo 'Dodawanie danych';
}
elseif($_GET['act'] == 'edytuj')
{
echo 'Edycja danych';
}
elseif($_GET['act'] == 'usun')
{
echo 'Usuwanie danych';
}
else
{
echo 'Wyświetlanie danych';
}

?>
```

Rozwiązanie to nie jest wygodne nie tylko ze względu na objętość takiego kodu i brak czytelności. Przypuśćmy, że z jakiegoś powodu musimy zmienić miejsce, z którego pobieramy informację o akcji. Trzeba to zrobić w czterech miejscach, a tych może być w teorii jeszcze więcej. Znacznie łatwiejszą w użyciu, a przy tym **wydajniejszą** alternatywą jest instrukcja switch. Działa ona w ten sposób, że wybiera spośród dostępnego zbioru określoną wartość na podstawie wartości pewnego wyrażenia i wykonuje zdefiniowany dla niej kod. Przepiszmy raz jeszcze powyższy przykład:

```
<?php

switch($_GET['act'])
{
case 'dodaj':
echo 'Dodawanie danych';
```

```
break;
case 'edytuj':
echo 'Edycja danych';
break;
case 'usun':
echo 'Usuwanie danych';
break;
default:
echo 'Wyświetlenie danych';
}

?>
```

W nawiasie polecenia **switch** definiujemy, jakiemu wyrażeniu pragniemy sprawdzić wartość. Wewnątrz nawiasów klamrowych używamy struktury **case wartość:**, aby zdefiniować dopuszczalne wartości, czyli stany wyrażenia. Po dwukropku piszemy odpowiedni kod. Jeżeli żaden ze stanów nie spełnia naszych oczekiwań, istnieje także klauzula **default:** pisana na samym końcu opisująca domyślne zachowanie. Jej już nie musimy dodawać, jeżeli tego nie potrzebujemy, niemniej często się ona przydaje. Zwróć uwagę na komendę **break;** stojącą na końcu każdego kodu przypisanego do **case**. Mówi ona, że wykonywanie przerywane jest w tym miejscu i PHP ma skoczyć do końca switcha, a nie przypadkiem wykonać kolejny stan. Taka oryginalna budowa wynika z jednego prostego powodu: jeżeli chcemy dla trzech różnych stanów wykonać to samo zadanie, po prostu piszemy pod rząd trzy case'y, potem kod i na końcu przerwanie. W powyższym przykładzie dodaliśmy alternatywną nazwę pierwszej akcji: "dod". Kod po przeróbkach wygląda tak:

```
<?php

switch($_GET['act'])
{
case 'dod':
echo 'Jak nie damy komendy "break", to pokaże nam się też...<br/>';
case 'dodaj':
echo 'Dodawanie danych';
break;
case 'edytuj':
echo 'Edycja danych';
break;
case 'usun':
echo 'Usuwanie danych';
break;
default:
echo 'Wyświetlenie danych';
}

?>
```

Wywołując skrypt jako *nazwapliku.php?act=dodaj*, zobaczymy:

Dodawanie danych

Wywołując skrypt jako *nazwapliku.php?act=dod*, zobaczymy:

Jak nie damy komendy "break", to pokaże nam się też...
Dodawanie danych

PHP wykonał zarówno stan "dod", jak i następujący po nim "dodaj", gdyż w tym pierwszym brakowało komendy **break**.

Kod stanu może zawierać inne struktury kontrolne:

```
<?php

switch($_GET['act'])
{
case 'dodaj':
if($_SERVER['REQUEST_METHOD'] == 'POST')
{
echo 'Dodawanie danych';
}
else
{
echo 'Formularz dodawania';
}
break;
case 'edytuj':
echo 'Edycja danych';
break;
case 'usun':
echo 'Usuwanie danych';
break;
default:
echo 'Wyświetlenie danych';
}

?>
```

Tutaj dodaliśmy instrukcję if, która sprawdza, czy żądanie nadeszło do nas z formularza HTTP, który należy przetworzyć, czy normalnie (wtedy wyświetlamy formularz). *$_SERVER['REQUEST_METHOD']* zawiera nazwę metody, za pomocą której odbyło się żądanie HTTP.

Instrukcję switch warto stosować, kiedy wybieramy konkretną możliwość z określonego w kodzie zbioru. Dla PHP zaletą jest, że "wie", jakie są wszystkie stany. Instrukcja warunkowa if jest bardziej ogólna i tam interpreter sprawdza po prostu po kolei wszystkie warunki, aż nie natrafi na pasujący, nie zagłębiając się w jakieś większe zależności między danymi. Switcha nie należy używać, kiedy mamy tylko dwa stany, gdyż takie zagranie przypominałoby wytoczenie haubicy do zabicia komara. Instrukcja ta nie sprawdza się także przy wszystkich bardziej ogólnych warunkach rodzaju "mniejszy, większy".

Instrukcja for

Pętle

Wszystkie kolejne struktury kontrolne, jakie poznamy, określa się jednym wspólnym terminem: *pętle*.

Pętlą nazywamy strukturę kontrolną powtarzającą dany kod do czasu spełnienia określonego warunku.

Wiemy już, że pętla powtarza w kółko pewien fragment kodu. Różnice między poszczególnymi rodzajami dotyczą tego, jak i kiedy jest ona przerywana. Na początek zajmiemy się pętlą for. Pokazuje ona pazurki, kiedy zliczamy ilość wywołań pętli i na podstawie tego określamy, czy trzeba ją przerwać, czy nie. W for definiujemy trzy wyrażenia:

- Startu - najczęściej inicjuje licznik wywołań
- Końca - warunek zakończenia
- Iteracji - najczęściej zwiększa licznik wywołań

Oddzielone są one średnikami. Pokażemy to na przykładzie skryptu wyświetlającego liczby od 0 do 9.

```
<?php
```

```
for($i = ; $i < 10; $i++)
{
echo $i.'<br/>';
}

?>
```

Warunek startu tworzy nową zmienną *$i* z wartością zero. Następnie określamy, że dopóki *$i* jest mniejsze od 10, pętla ma się powtarzać. Przy każdym cyklu należy zwiększyć wartość *$i* o 1.

Uwaga!
Uważaj na warunek końca pętli! Jeżeli niepoprawnie go zdefiniujesz, pętla może nie wykonać się wcale albo też powtarzać się w nieskończoność. Drugi przypadek nie jest aż taki groźny, ponieważ PHP automatycznie przerywa wykonywanie skryptu, jeżeli trwa ono ponad 30 sekund.

Proste wyświetlanie tablic

Pętla for jest użyteczna przy wyświetlaniu tablic z indeksami numerycznymi. Mamy plik tekstowy z zawartością:

```
Litwo, ojczyzno moja! Ty jesteś jak zdrowie,
Ile cię trzeba cenić, ten tylko się dowie,
Kto cię stracił, dziś piękność twą w całej ozdobie
Widzę i opisuję, bo tęsknię po tobie.
```

Zastosujemy funkcję *file()*, aby wczytać go do pamięci z jednoczesnym rozbiciem na poszczególne wiersze zapisane w tablicy. W ten sposób będziemy je mogli wyświetlić jako elementy listy wypunktowanej:

```
<?php

$plik = file('plik.txt');

echo '<ul>';
for($i = , $x = count($plik); $i < $x; $i++)
{
echo '<li>'.trim($plik[$i]).'</li>';
}
echo '</ul>';

?>
```

Do określenia ilości wierszy użyliśmy poznanej już wcześniej funkcji *count()*. Przy wyświetlaniu stosujemy jeszcze jedną: *trim()*. Usuwa ona z początku i końca każdego wiersza białe znaki, tj. spacje, zejścia do nowej linii, tabulatory. Wynikiem działania skryptu jest zawartość pliku wyświetlona w liście wypunktowanej.

Zwróć uwagę na specyficzną budowę wyrażenia inicjacji pętli. Pragniemy utworzyć dwie zmienne, dlatego oddzielamy je przecinkami. Podobną sztuczkę możemy zastosować również w wyrażeniu iteracyjnym. Można się zapytać, dlaczego zastosowaliśmy tak rozbudowaną konstrukcję. Przecież dopuszczalne jest także napisanie:

```
for($i = ; $i < count($plik); $i++)
```

W typowych sytuacjach obie konstrukcje zachowają się podobnie, lecz warto pamiętać o pewnym niuansie technicznym. Pierwsza z konstrukcji pobiera ilość elementów tablicy na samym początku. Jeżeli któryś cykl pętli doda jakiś element, nie zostanie on uwzględniony. W drugim przypadku ilość ta jest pobierana po każdym cyklu, zatem pętla dysponuje bieżącymi informacjami o wielkości tablicy i wszelka jej zmiana zostanie uwzględniona w ilości wykonanych cykli. Sposób ten jest jednak mniej wydajny od pierwszego.

Break i Continue

Przy okazji omawiania instrukcji switch poznaliśmy komendę **break**. Ma ona bardzo duże zastosowanie przy

pętlach, które potrafi przerywać. Istnieje także kolejne polecenie: **continue**. Przerywa ono jedynie aktualny cykl pętli i powoduje rozpoczęcie następnego.

Mamy prosty ciąg tekstowy:

Komenda; Komenda; Komenda; Komenda. To już pomijamy.

Wiemy o nim trzy rzeczy:

1. Spacje ignorujemy
2. Kropka oznacza koniec wprowadzania komend
3. Średnik separuje komendy

Naszym zadaniem jest wprowadzenie komend do tablicy, aby można je było łatwiej przetwarzać. Skrypt ten będziemy pisać kawałek po kawałku. Na początek stwórzmy sobie parę zmiennych:

```
<?php

$tekst = 'Komenda; Komenda; Komenda; Komenda. To już pomijamy.';
$tablica = array( => '');
$t = ;
```

$tekst to tekst do przetworzenia. *$tablica* jest miejscem docelowym komend z "firmowo" utworzonym pierwszym pustym elementem. *$t* to licznik mówiący, do którego elementu tablicy wprowadzamy znaki.

Rozpoczynamy pętlę. Do pobrania długości ciągu użyjemy funkcji *strlen()*. *$i* to licznik położenia w ciągu tekstowym. Wskazuje na aktualnie przetwarzany znak:

```
  for($i = ; $i < strlen($tekst); $i++)
{
```

Implementujemy możliwość pierwszą. Spacje ignorujemy, dlatego przy ich napotkaniu przerywamy aktualny cykl pętli komendą **continue** i przechodzimy do następnego:

```
        if($tekst{$i} == ' ')
{
continue;
}
```

Zauważ, jak odwołujemy się do określonego znaku wewnątrz ciągu: *$tekst{$i}*. Numer znaku (począwszy od zera) podajemy jako indeks w nawiasach klamrowych.

Druga możliwość - po napotkaniu kropki przerwać pętlę wcześniej:

```
        if($tekst{$i} == '.')
{
break;
}
```

Przechodzimy do ewentualności trzeciej. Przy średniku należy przesunąć się na nowy element tablicy wynikowej i zainicjować go pustym ciągiem. Każdy inny znak wprowadzamy do aktualnego elementu tablicy:

```
        if($tekst{$i} == ';')
{
$t++;
$tablica[$t] = '';
}
else
{
$tablica[$t] .= $tekst{$i};
```

```
}
```

Teraz dopełnienie formalności, tj. zamknięcie pętli i wyświetlenie zawartości tablicy funkcją *print_r()*:

```
  }

echo '<pre>';
print_r($tablica);
echo '</pre>';
?>
```

Zapytajmy się, jak przerwać pętlę, jeżeli jesteśmy aktualnie w instrukcji switch? Wywołanie **break** i **continue** będzie się przecież odnosiło do niej, a tego nie chcemy. Rozwiązaniem jest podanie po nich numeru określającego, której instrukcji wzwyż dotyczy wywołanie. Przepiszmy jeszcze raz powyższy kod z wykorzystaniem instrukcji wyboru (notabene nawet bardziej pasującej w tym przypadku):

```
<?php

$tekst = 'Komenda; Komenda; Komenda; Komenda. To już pomijamy.';
$tablica = array( => '');
$t = ;

for($i = ; $i < strlen($tekst); $i++)
{
switch($tekst{$i})
{
case ' ':
continue 2;
case '.':
break 2;
case ';':
$t++;
$tablica[$t] = '';
break;
default:
$tablica[$t] .= $tekst{$i};
}
}

echo '<pre>';
print_r($tablica);
echo '</pre>';
?>
```

Przy stanach spacji oraz kropki wywołujemy komendy **continue** oraz **break** z parametrem 2, aby podkreślić, że dotyczą one pętli for, a nie instrukcji switch. **break** w kodzie obsługi średnika nie ma parametru, więc odnosi się do instrukcji switch.

Instrukcja while

Kolejną pętlą jest while, będąca znacznie prostszą odmianą poznanej ostatnio pętli for. Wymagany jest tu jedynie warunek zakończenia, a pętla wykonuje się, dopóki jest on prawdziwy. Oto prosty przykład:

```
<?php

while(rand(,10) != 8)
{
echo 'Jeszcze nie wylosowałem!<br/>';
}
```

```
?>
```

Pętla ta będzie wykonywała się, dopóki funkcja *rand()* nie wylosuje liczby 8. Jeżeli dostaniemy ją już w pierwszym sprawdzeniu, napis nie pojawi się w ogóle.

Ze względu na taką ogólną konstrukcję while przydaje się tam, gdzie musimy coś powtarzać do czasu osiągnięcia pewnego stanu. Sztandarowym przykładem jest czytanie pliku, gdzie takim specyficznym stanem, w którym musimy przerwać naszą pracę, jest jego koniec. Zastosowanie pętli while będzie tu o wiele lepsze, niż obliczanie na podstawie wielkości pliku, ile "segmentów" musimy pobrać i zabawa z licznikami.

```
<?php
$f = fopen('plik.txt', 'r'); // 1
while(!feof($f)) // 2
	{
echo fgets($f, 16); // 3
	}

fclose($f); // 4
 ?>
```

Opis działania:

1. Otwieramy plik do odczytu. Uchwyt do niego zapisujemy w zmiennej *$f*. W ten sposób poznaliśmy nowy typ danych: *Resource*, czyli zasób.
2. Dopóki nie osiągniemy końca pliku...
3. Pobieraj z niego kolejne 16-znakowe bloki.
4. Na koniec zamykamy połączenie z plikiem.

Pętlę while można przerobić na pętlę for bez większych trudności. Oto nowa wersja pierwszego przykładu z rozdziału poprzedniego:

```
<?php
$i = ;
while($i < 10)
{
echo $i.'<br/>';
$i++;
}
?>
```

Tu także można stosować komendy *break* oraz *continue* poznane w poprzednim rozdziale.

Pętla while przyda się nam, gdy zaczniemy omawiać komunikację z bazami danych. Zostanie tam wykorzystana do pobierania rekordów.

Instrukcja do while

W przeciwieństwie od normalnego while, tutaj warunek sprawdzany jest na końcu, tak więc pętla zostanie wykonana przynajmniej raz. Nie jest to często wykorzystywana właściwość, ale warto o niej pamiętać.

Składnia pętli do while jest dość specyficzna. Przed nawiasem klamrowym pojawia się jedynie słowo kluczowe **do**, a **while** z warunkiem znajduje się na samym końcu. Przedstawimy to na przykładzie takiego oto skryptu:

```
<?php
do
{
echo "Podaj i: \n";
```

```
fscanf(STDIN, "%d\n", $i); // 1
	}
while($i < 10);
?>
```

Nie będziemy uruchamiali go w przeglądarce, ale **w linii komend**. Powyższy skrypt pracuje w konsoli systemowej i może pobierać stamtąd dane poprzez wiersz zaznaczony jako **1** (nie przejmuj się, że nie rozumiesz jego budowy. Poznamy ją dalej). Aby uruchomić skrypt, uruchom konsolę i ustaw się poleceniem *cd* na katalogu, w którym zainstalowałeś PHP. Następnie wydaj następujące polecenie:

```
php -f /scieżka/do/skrypt.php
```

Jako wartość parametru *-f* musisz podać pełną ścieżkę do skryptu, który chcesz uruchomić.

Zauważ, że dzięki użyciu pętli do while, nie musimy umieszczać w skrypcie dwa razy kodu do pytania się o zmienną *$i*. Oto analogiczny kod z wykorzystaniem normalnego while:

```
<?php
echo "Podaj i: \n";
fscanf(STDIN, "%d\n", $i);

while($i < 10)
{
echo "Podaj i: \n";
fscanf(STDIN, "%d\n", $i);
}
?>
```

Tutaj musimy powielić kod dwa razy, bo przecież przed sprawdzeniem warunku wypada się choć raz zapytać użytkownika, co należy sprawdzić. W poprzednim przykładzie mogliśmy do tego celu użyć kodu wewnątrz pętli, ponieważ mieliśmy zagwarantowane wykonanie jej kodu przynajmniej raz.

Instrukcja foreach

Ostatnią pętlą jest foreach. Ma ona specyficzne zastosowanie, ponieważ służy wyłącznie do przeglądania zawartości typów złożonych: tablic oraz obiektów. Kod wewnątrz niej jest powtarzany dla każdego z elementów tablicy, a on sam jest na ten czas przenoszony do tworzonej przez pętlę zmiennej. Wróćmy do naszego przykładu z pętlą for odczytującego zawartość pliku. Przepiszemy go z wykorzystaniem foreach:

```
<?php
$plik = file('plik.txt');

echo '<ul>';
foreach($plik as $linia)
{
echo '<li>'.trim($linia).'</li>';
}
echo '</ul>';

?>
```

Teraz skrypt ma o wiele bardziej przejrzystą budowę. Przyjrzyjmy się deklaracji pętli:

```
foreach($plik as $linia)
```

Mówi nam ona, że pętla ma analizować tablicę *$plik*, a aktualnie przetwarzany element ma być zapisany w zmiennej *$linia*.

Foreach umożliwia nam także zwracanie nazw indeksów elementów:

```
<?php
$plik = file('plik.txt');

echo '<ul>';
foreach($plik as $numer => $linia)
{
echo '<li>Linia #'.$numer.': '.trim($linia).'</li>';
}
echo '</ul>';

?>
```

Foreach ma tę przewagę nad innymi pętlami, że wie, jakie elementy należą do tablicy i zawsze przetworzy tylko je. Gdybyśmy przed wyświetleniem pliku usunęli z niego np. linijkę 1, pętla for nie dałaby rady, próbując przetworzyć nieistniejący element. Nie robi oczywiście tego dla złośliwości, lecz dlatego, że operuje na liczniku i nie wie, do czego jest on przez nas dalej wykorzystywany.

```
<?php
$plik = file('plik.txt');
unset($plik[1]); // usuwamy linijkę o indeksie 1
	echo '<ul>';
foreach($plik as $numer => $linia)
{
echo '<li>Linia #'.$numer.': '.trim($linia).'</li>';
}
echo '</ul>';

?>
```

Tworzone przez foreach zmienne są jedynie kopiami oryginalnych wartości, dlatego próba ich modyfikacji wewnątrz pętli w żaden sposób nie wpłynie na zawartość tablicy:

```
<?php
$plik = file('plik.txt');

foreach($plik as $linia)
{
$linia = 'Próba skasowania';
}

echo '<pre>';
print_r($plik);
echo '</pre>';
?>
<?php
$plik = file('plik.txt');

foreach($plik as &$linia)
{
$linia = 'Próba skasowania';
}

echo '<pre>';
print_r($plik);
echo '</pre>';
?>
```

Wewnątrz pętli próbujemy przypisać wartość do zmiennej *$linia*. Owszem, udaje nam się to, ale nowa treść nie

trafia w ogóle do tablicy i systemowe wyświetlenie jej zawartości ukazuje brak jakiejkolwiek reakcji. Czy zatem możliwe jest dokonywanie przypisań wewnątrz foreach? Oczywiście. Są dwie sztuczki. Pierwsza polega na wykorzystaniu zwracanego przez pętlę indeksu. Pousuwajmy z tablicy zbędne białe znaki:

```
<?php
$plik = file('plik.txt');

foreach($plik as $i => $linia)
{
// jeszcze jakiś napis sobie doklejmy
            $plik[$i] = trim($linia).' [OK]';
}

echo '<pre>';
print_r($plik);
echo '</pre>';
?>
```

Rozwiązanie to jest nieco trikowe, ale działa. Możemy jednak zastosować coś innego. PHP posiada pewien element zwany referencją. Ogólnie rzecz biorąc jest to odnośnik do zmiennej, który zachowuje się tak, jak ona. Modyfikacja referencji powoduje także modyfikację oryginalnego elementu. Począwszy od PHP 5, referencje można używać w pętli foreach. Wystarczy poprzedzić w jej deklaracji zmienną *$linia* znakiem **&**:

```
<?php
$plik = file('plik.txt');

foreach($plik as &$linia)
{
$linia = trim($linia).' [OK]';
}

echo '<pre>';
print_r($plik);
echo '</pre>';
?>
```

Teraz modyfikacja zmiennej *$linia* jest równoznaczna z modyfikacją aktualnego elementu w tablicy *$plik*, ponieważ zmienna jest takim właśnie odnośnikiem. O referencjach szerzej powiemy w rozdziale **Inne elementy składni**.

Porada
Jeżeli elementy twojej tablicy są bardzo duże i liczą np. po kilka kilobajtów, użyj referencji do zwiększenia wydajności. Normalnie musiałyby być one w całości kopiowane, co tylko pochłaniałoby zbędnie moc. Referencja natomiast utworzy do nich zwyczajny odnośnik i czasochłonne kopiowanie nie będzie miało miejsca.