Warunki i pętle

if – instrukcje warunkowe

Często potrzebujemy sprawdzić, czy zmienna jest większa lub mniejsza od danej wartości. W zależności czy warunek jest spełniony wykonujemy osobny skrypt. Przykładowo użytkownik wchodzi na stronę przeznaczona dla osób pełnoletnich. W zależności od tego czy potwierdzi dorosłość trzeba go albo przekierować na stronę główną albo wyprosić.

Instrukcja warunkowa if
Powyżej opisany przypadek w prosty sposób rozwiąże zastosowanie instrukcji if. Jej składnia jest następująca:

```
<?php
	if (warunek) // w nawiasie podajemy warunek do sprawdzenia
	{
		instrukcje
	}
?>
```

Sprawdza ona, czy warunek podany w nawiasie został spełniony. Warunki w argumencie można łączyć za pomocą operatorów logicznych, poznanych w poprzedniej lekcji. Całkowita wartość nawiasu sprowadza się do określenia wartości logicznej – TRUE lub FALSE. Jeśli warunek jest spełniony (wartość TRUE) instrukcje zostaną wykonane. Jeśli natomiast nie jest (wartość FALSE) instrukcje zostaną pominięte.

Przykład zastosowania.
Napiszemy prosty kod, który wyświetli komunikat o parzystości liczby. W zależności czy wartość zmiennej będzie parzysta, czy też nie, wykonana zostanie inna instrukcja.

```
<?php
	$a = 7; // przypisujemy wartość zmiennej $a
	if ($a%2 > 0) // reszta z dzielenia przez 2
	{
		echo "Liczba nieparzysta";
	}
	if ($a%2 == 0) // brak reszty z dzielenia przez 2
	{
		echo "Liczba parzysta";
	}
?>
```

W skrócie… Jeśli interpreter natrafia na warunek **if**, sprawdza wartość logiczną w nawiasie. W przypadku zwrócenia TRUE, wykona się instrukcja w nawiasach klamrowych. Jeżeli natomiast zwrócona zostanie wartość FALSE, kod w nawiasach zostanie pominięty. W przypadku, gdy mamy tylko jedną instrukcję, nie musimy stosować nawiasów klamrowych. Poprawny będzie również zapis:

```
<?php
	$a = 7; // przypisujemy wartość zmiennej $a
	if ($a%2 > 0) // reszta z dzielenia przez 2
		echo "Liczba nieparzysta";
	if ($a%2 == 0) // brak reszty z dzielenia przez 2
		echo "Liczba parzysta";
?>
```

Istotnym elementem instrukcji warunkowych jest składanie warunku w nawiasie z wielu pojedynczych warunków. Możesz zestawiać ze sobą różne warunki i łączyć je na zasadach: TO I TO oraz TO LUB TO. Zobacz poniższe przykłady:

```
<?php
	$a = 7; // przypisujemy wartość zmiennej $a
	if ($a > 0 && $a < 10) // $a jest większa niż 0 I jest mniejsza niż 10 - oba warunki muszą zostać spełnione
		echo "Liczba między 0 a 10";

	if ($a < 0 || $a > 10) // $a jest mniejsza niż 0 LUB jest większa niż 10 - jeden z warunków musi zostać spełniony
		echo "Liczba mniejsza od 0 lub większa niż 10";
?>
```

else if

Poprzednio omawialiśmy przypadek parzystości liczby. Co w przypadku, gdy chcemy rozważyć kilka alternatyw? Wyobraźmy sobie, że potrzebujemy sprawdzić, czy liczba jest podzielna przez osiem. Jeśli nie jest, chcemy sprawdzić, czy dzieli się przez cztery. Jeśli nie, to przez dwa. Jeżeli żaden warunek nie zostanie spełniony wyświetlamy odpowiedni komunikat. Liczba podzielna przez osiem jest podzielna również przez cztery i przez dwa. Jeśli użylibyśmy czterech instrukcji **if**, trafiając na liczbę osiem spełnione byłyby trzy warunki. My chcemy, aby tylko jedna instrukcja została wykonana. W PHP możemy to osiągnąć poprzez użycie dodatkowych alternatyw **elseif**.

Zobaczmy przykład poniżej:

```
<?php
	$a = 34; // przypisujemy wartość zmiennej $a

	if ($a%8 == 0) // liczba podzielna przez osiem
		echo "Liczba podzielna przez osiem";

	elseif ($a%4 == 0) // liczba podzielna przez cztery
		 echo "Liczba podzielna przez 4, ale nie przez 8";

	elseif ($a%2 == 0) // liczba podzielna przez dwa
		echo "Liczba podzielna przez 2, ale nie przez 4";
```

```
	else // żadna z powyższych
		echo "Liczba nieparzysta";
?>
```

Jak widać poprzez zastosowanie **elseif**, możemy w jednym podejściu rozważyć kilka możliwości. Gdy jedna z nich zostanie spełniona, pozostałe nie będą już sprawdzane. Podobnie jak przy zwykłym **if**, możemy pominąć klamrowe nawiasy w przypadku tylko jednej instrukcji. Podsumowując, dzięki **elseif** możemy dołożyć kolejny warunek do sprawdzenia w przypadku, gdy poprzedni nie został spełniony. Jeżeli którykolwiek z poprzedzających warunków został spełniony, kolejne bloki **elseif** nie zostaną sprawdzone ani wykonane.

Warunek switch

W poprzedniej lekcji poznaliśmy budowanie kilku alternatyw za pomocą instrukcji **elseif**. Podobny efekt uzyskujemy stosując instrukcję warunkową **switch**.
Jest to popularny warunek, obecny niemal we wszystkich językach programowania. W niektórych zamiast **switch… case…** stosuje się składnie **case… when…**, lecz zasada działania jest identyczna.

Struktura PHP Switch wygląda następująco:

```
<?php
$a = 72; // przypisujemy wartość zmiennej $a

	switch ($a) // sprawdzamy zmienną $a
	{
		case 1:
			echo "Wartość zmiennej a to 1";
			break;

		case 2:
			echo "Wartość zmiennej a to 2";
			break;

		case 3:
			echo "Wartość zmiennej a to 3";
			break;

		case 72:
			echo "Wartość zmiennej a to 72";
			break;

		default:
			echo "Żadna z powyższych";
			break;
}
?>
```

Jako argument podajemy pewne wyrażenie, np. Jakąś zmienną. Następnie rozważamy je pod pewnymi warunkami, stosując słowa kluczowe **case**. Jeśli żaden z **case**'ów nie zwróci wartości

true, wywoływana jest instrukcja po słowie **default**. Jest to instrukcja domyślna, której użycie jest opcjonalne. Jeśli jej nie zastosujemy, a żaden **case** nie zwróci prawdy, **switch** nie zrobi nic.

Break
Bardzo istotnym elementem w poprawnym konstruowaniu **switch'a** są słówka **break**. Powiemy sobie o nim więcej przy okazji omawiania pętli. Powoduje ono mniej więcej efekt wyskoczenia z obecnej instrukcji i przejścia na jej koniec. Na ten moment musisz jedynie zapamiętać, że **break** jest niezbędny po zakończeniu pisania instrukcji dla danego przypadku.

Kiedy switch a kiedy if else

Na to pytanie nie ma jednoznacznej odpowiedzi. Co kto lubi.
Generalnie, jeżeli warunków jest dużo, znacznie lepiej wygląda instrukcja **switch**.

Pętla While

Czym są pętle?

Wyobraźmy sobie sytuację, że chcemy wyświetlić ciąg liczb, który zaczyna się od pewnej wartości przechowywanej w zmiennej, a kończy na 100.
Należałoby za każdym razem sprawdzać czy 100 już zostało osiągnięte. Uczyliśmy się na poprzednich lekcjach, że sprawdzaniem warunków zajmują się takie instrukcje jak **if**, czy **switch**.
Z ich wykorzystaniem wyglądałoby to mniej więcej tak:

```
<?php
	if($zmienna < 101)
	{
		echo $zmienna;
		$zmienna++;
	}
	if($zmienna < 101)
	{
		echo $zmienna;
		$zmienna++;
	}
	.
	.
	.
	if($zmienna < 101)
	{
		echo $zmienna;
		$zmienna++;
	}
?>
```

Jak widzisz, zajmuje to dość dużo miejsca. Nie wiemy w dodatku, ile instrukcji warunkowych **if** musimy wstawić, aby dojść do 100, bo nie znamy pierwszej zmiennej. Potrzebna jest nam metoda, która będzie wykonywała daną instrukcję, aż warunek nie zostanie osiągnięty, niezależnie od ilości przebiegów. Taką metodę w programowaniu nazywamy pętlą.

Pętla while

Zobaczmy rozwiązanie powyższego problemu przy pomocy pętli while.
Składnia pętli **while** w PHP wygląda następująco:

```
<?php
	while($zmienna < 101) // warunek kontynuacji pętli
	{
		echo $zmienna;
		$zmienna++;
	}
?>
```

Widzimy, że kod stał się dużo krótszy. Do tego nie obchodzi nas początkowa wartość **$zmiennej**, gdyż nie określamy liczby przebiegów pętli.
While z angielskiego tłumaczymy jako „podczas gdy". Wyżej przedstawiony warunek możemy przetłumaczyć na język polski: „Podczas gdy zmienna jest mniejsza od 101 wykonaj następujące instrukcje".
Jeśli zmienna osiągnie wartość 101, interpreter przechodzi do pozostałej części kodu. Jeżeli **$zmienna** na początku byłaby większa od 100, instrukcje w pętli nie wykonałyby się ani razu.
Podsumowując, pętle to takie konstrukcje w programowaniu, które wykonują wybrany blok kodu do momentu, aż warunek im podany nie zostanie złamany. Dopóki warunek jest spełniony, pętla wykonuje blok operacji i znowu sprawdza, czy warunek jest spełniony.
Bez pętli niemal nie da się programować. Dlatego ważne, żeby dobrze zrozumieć ich działanie. Dlatego zobaczmy jeszcze dwa przykłady:

```
<?php
	while($zmienna < 101) // warunek kontynuacji pętli
	{
		 echo $zmienna;
		$zmienna += 10; // zmienna może się zmieniać szybciej/wolniej/wcale
	}
?>

<?php
	while($zmienna < 101 && $inna > 100) // warunek kontynuacji pętli
	{
		echo $zmienna;
		echo $inna;
		$zmienna += 10; // zmienna może się zmieniać szybciej/wolniej/wcale
		$inna -= 5; // sprawdzanych może być wiele zmiennych
	}
?>
```

Zauważ, że warunek kontynuacji pętli (wykonywania kodu w bloku nawiasów klamrowych) może składać się z kilku warunków. Podobnie jak przy instrukcjach warunkowych, możemy łączyć wiele warunków i dostawać logiczny rezultat.

Pętla do… while

PHP **do while** jest odpowiednikiem pętli **php while**, który działa na tej samej zasadzie. Jedyna różnica tkwi w momencie sprawdzania warunku. Podczas gdy **while** robił to zaraz na początku, **do… while** dokonuje sprawdzania po zakończeniu instrukcji. Wynika z tego fakt, że niezależnie, czy warunek zwróci **true** czy **false**, pętla wykona się przynajmniej jeden raz.
Są przypadki, że pozwala to uniknąć duplikowania kodu, który byłby nieunikniony w przypadku korzystania wyłącznie z pętli **while**. Podkreślę raz jeszcze – pętli **do while** używamy wtedy, gdy chcemy wykonać instrukcję **przynajmniej raz** – niezależnie od spełnienia warunku.

Struktura pętli **do while** w PHP wygląda następująco:

```
<?php
	$zmienna = 200;
	do // instrukcje do wykonania
	{
		echo $zmienna;
		$zmienna++;
	}
	while($zmienna < 101) // warunek kontynuacji pętli
?>
```

Powyższy przykład odwołuje się do problemu przedstawionego w poprzedniej lekcji. Jak widać, w tym przypadku pętla **do… while** nie spełnia zamierzonego zadania. Nieważne, czy **$zmienna** jest większa czy mniejsza od 101, zmienna zostanie wyświetlona przynajmniej raz. Z racji, że chcieliśmy wyświetlić ciąg liczb do 100, pokazanie liczby większej niż 100 jest niepożądane.
Są jednak przypadki, kiedy wygodniej jest użyć **do… while**, zamiast **while**.
W programowaniu nie ma jednej, wyznaczonej, najlepszej drogi. Często jedno zadanie można wykonać przy pomocy różnych technik. Tak naprawdę każdą pętlę można zastąpić inną, zmieniając nieco warunki i przebudowując program. Przy każdej otrzymamy zamierzony efekt. Wszystko sprowadza się do zrozumienia sposobu działania i inteligentnego użycia nabytej wiedzy.
Na koniec ćwiczenie. Chcemy wypisać wszystkie liczby mniejsze od **$zmienna**, ale większe od 0. Z tym dodatkowym warunkiem, że liczba **$zmienna ma być zawsze wyświetlona** na początku (niezależnie czy jest dodatnia czy ujemna). Przykładowa realizacja z pętlą **do while**:

```
<?php
	$zmienna = -10;
	do // instrukcje do wykonania
	{
		echo $zmienna;
		$zmienna--;
	}
	while($zmienna > 0) // warunek kontynuacji pętli
?>
```

W tym przypadku, nawet gdy zmienna wynosi -10, jej wartość zostanie wyświetlona. Gdy kod dojdzie do warunku, nie będzie on spełniony. Jednak pierwszy przebieg się wykona i na ekranie zobaczymy -10.

Pętla FOR

Ostatnią omawianą przez nas pętlą będzie pętla **for**. Jest bardzo popularna w każdym języku programowania.
Stosujemy ją wtedy, gdy wiemy z góry, ile przebiegów pętla ma odbyć. Jeśli np. wiemy, że kod chcemy wykonać dokładnie sto razy, z pomocą przychodzi nam pętla for.
Jej konstrukcja jest następująca:

```
<?php
	for($i=0; $i < 100; $i++)
	{
		// instrukcje do wykonania
		// z każdą iteracją
	}
?>
```

Pierwszym elementem w nawiasie jest przypisanie zmiennej iteracyjnej początkowej wartości. To ona decyduje o spełnieniu warunku pętli, gdyż jej wartość porównuje się z drugim elementem. Z każdym przebiegiem pętli wartość zmiennej **$i** zmienia się, aż nie spełni warunku końcowego.
Warunek końcowy to drugi element w nawiasie. To właśnie z nim porównujemy zmienną $i. Jeśli warunek jest spełniony, blok kodu wykona się. Jeśli nie, wyjdzie z pętli. Mówiąc prostymi słowami, „wykonuj blok pętli tak długo, dopóki warunek jest prawdziwy". Z przykładu powyżej, dopóki zmienna **$i** jest mniejsza niż 100, dopóty wykonuj kod.
Trzecim elementem wskazujemy, jak ma przebiegać zmiana wartości zmiennej **$i**. Zazwyczaj jest to inkrementacja lub dekrementacja (zwiększanie lub zmniejszanie wartości o 1). Nie jest to jednak koniecznością. Możemy ustalić, że nasza zmienna ma się zmieniać o 2 co każdy przebieg. Zobacz przykład poniżej:

```
<?php
	For ($i = 0; $i < 100; $i += 2)
	{
		// instrukcje do wykonania zwiększając $i o 2
	// z każdą iteracją
	}
?>
```

Nic nie stoi na przeszkodzie, by zmniejszać wartość **$i** z każdym przebiegiem. Możemy odliczać od 100 do 0, zamiast od 0 do 100. Ilość przebiegów będzie taka sama. Liczy się czytelność kodu i logika. Zobacz przykład ze zmniejszaniem wartości **$i** o 2 w każdej iteracji:

```
<?php
	For ($i = 100; $i > 0 ; $i -= 2)
	{
		// instrukcje do wykonania zmniejszając $i o 2
		// z każdą iteracją
	}
?>
```

Zasada działania pętli for

Pętla będzie wykonywać się tak długo, aż warunek nie zostanie spełniony. Brzmi to dość podobnie do działania pętli **While**. Różnica jednak polega na tym, że w **While** modyfikowaliśmy zmienną warunkową wewnątrz instrukcji, natomiast w **for** deklarujemy przebieg w nagłówku pętli. Dodatkowo w nagłówku przypisujemy początkową wartość zmiennej.

Poniższy przykład pokazuje, że pętle **for** i **while** są równoważne. Różnią się jedynie zapisem:

```
<?php
	// użycie pętli for
	For ($i=0;$i<10;$i++)
	{
		// instrukcje
	}

	// ten sam efekt z użyciem funkcji while
	$i=0
	while($i < 10)
	{
		// instrukcje
		$i++;
	}
?>
```

Każdą pętlę możemy zapisać za pomocą innej. Chodzi jedynie o przejrzystość kodu. Dlatego, gdy z góry znamy liczbę przebiegów, zazwyczaj stosujemy pętlę **for**. Jest to zrozumiałe dla każdego programisty.

Operator ?

Na początku rozdziału omówiliśmy zasadę działania instrukcji warunkowej **if** oraz jej rozszerzenie – **else**. Pokażę teraz, jak zastosować podobną konstrukcję z użyciem **operatora ” ? „**. Jest ona bardzo przydatna przy budowaniu krótkich instrukcji, mając proste warunki.

Zobaczmy, jak wygląda jej struktura:

```
<?php
	$a = 5; // przypisujemy wartość zmiennej $a
	$odpowiedz = ($a>5) ? 'Większa od 5' : 'Mniejsza, bądź równa 5';

	echo $odpowiedz;
?>
```

Czas na krótkie wyjaśnienie. **$odpowiedz** jest zmienną, do której przypiszemy wynik zwracany przez **operator ?**. Wyrażenie w nawiasie oznacza nasz warunek (w tym przypadku sprawdzamy, czy **$a** jest większa od 5). Jeżeli jest, **$odpowiedz** przyjmuje wartość pierwszą, czyli ‚**Większa od 5’**. Na ekranie wyświetli się komunikat ‚**Większa od 5’**. Jeśli natomiast **$a** nie będzie większa od 5, **$odpowiedz** przyjmie wartość drugą (po dwukropku), wyświetlając ‚**Mniejsza, bądź równa 5’**.

Przypisywanie wyniku do zmiennej pomocniczej nie jest konieczne. Spójrzmy na poniższy przykład:

```
<?php

	$a = 5; // przypisujemy wartość zmiennej $a
	echo ($a>5) ? 'Większa od 5' : 'Mniejsza, bądź równa 5';

?>
```

Jak widać wyżej, możemy od razu wyświetlić wynik **operatora ?** funkcją **echo**.

Podsumowanie

Napiszemy skrypt, który wyświetli X ciągów liczb od zera do dwudziestu (gdzie X będzie wyznaczała zmienna **$ilosc**). Jeśli **$ilosc** będzie mniejsza od zera, wyświetlimy X ciągów od dwudziestu do zera (X oznaczymy jako „- $ilość"). Jeżeli **$ilosc** będzie równa 0, wyświetlimy komunikat o braku ciągów.
Jest to ćwiczenie praktyczne, prowadzone w celach edukacyjnych, dlatego postaramy się użyć jak największej liczby poznanych metod. Na początek sprawdzimy zmienną **$ilosc** za pomocą **operatora ?**. Następnie zagnieździmy w pętli **while** pętlę **for** (**for** wyświetli 20 liczb, a **while** będzie odpowiedzialna za ilość ciągów).

Przegląd rozwiązania

```
<?php

	$ilosc = X; // przypisujemy dowolną wartość zmiennej $ilosc

	// zmienna $kontynuacja sprawdza, czy $ilosc jest zerem
	$kontynuacja = ($ilosc == 0) ? 0 : 1;

	// jeśli nie jest, możemy kontynuować
	if ($kontynuacja == 1)
	{
	if ($ilosc > 0) // wyświetlamy ciągi od 0 do 20
		while ($ilosc > 0) // musimy wypisać $ilosc ciągów
		{
			for($i=0;$i<21;$i++) // 20 liczb za pomocą for
				echo $i;
			$ilosc--; // zmniejszamy, aż dojdzie do 0
			echo "<br/>"; // przejście do kolejnej linijki
		}
		else // $ilosc jest ujemna, wyswietlamy od 20 do 0
			while ($ilosc < 0) // wypisujemy -$ilosc ciągów
			{
				for($i=20;$i>=0;$i--) // 20 liczb za pomocą for
				echo $i;
				$ilosc++; // zwiększamy, aż dojdzie do 0
				echo "<br/>"; // przejście do kolejnej linijki
			}
	}
	else // jeśli kontynuacja wynosi 0
```

```
        echo "Brak ciągów liczb";

?>
```

Przeanalizujmy napisany przez nas kod. Na początku sprawdzamy, czy **$ilosc** równa się zero. Jeśli tak, instrukcje w warunku **if** nie zostaną wykonane. Interpreter przeskoczy do słowa **else**, którego instrukcja wypisze nam komunikat o braku ciągów. Jeśli jest różna od zera, sprawdzamy, czy jest od niego większa. Jeżeli jest, wykonujemy pętlę **while** dekrementując **$ilosc** tak długo, aż wyniesie ona zero. Jak łatwo zauważyć, pętla wykona się X razy, gdzie X to wartość **$ilosc**. Do wyświetlania dwudziestu jeden liczb użyta została pętla **for**. Na koniec **
**, żeby przełamywać kolejne ciągi.
W przypadku, gdy **$ilosc** jest mniejsza od zera, wykonają się instrukcje dla słowa kluczowego **else**. Podejście jest niemal identyczne. Różnica tkwi w warunku kontynuacji pętli, jak również w wyświetlaniu (liczby wypisujemy wspak). **$ilosc** jest teraz inkrementowana, gdyż musimy z liczby ujemnej dojść do zera. Pewnie zauważyliście, że w niektórych momentach nie ma nawiasów klamrowych po pętli lub warunku. Jeżeli wykonujemy tylko jedną instrukcję (również pętlę lub warunek) nawiasy nie są wymagane.

Ćwiczenia

- Napisz skrypt, który za pomocą dowolnie wybranych pętli wypisze tabliczkę mnożenia z liczbami od 1 do 10; będzie konieczne zagnieżdżenie jednej pętli w drugiej; podobna konstrukcja jak w przykładzie powyżej,
- zmodyfikuj tabliczkę mnożenia tak, żeby liczby parzyste kolorowało na niebiesko, a nieparzyste na zielono; użyj warunku if,
- wykorzystując instrukcję warunkową switch, napisz skrypt, który w zależności od wartości zmiennej (2, 3, 4) wyświetli ciąg dziesięciu kolejnych liczb podniesionych do potęgi o wartości zmiennej (2, 3, 4); skorzystaj z wiedzy, że x^3 to inaczej x*x*x (analogicznie x^2 i x^4)