Proměnné

198 Rozsah viditelnosti proměnných

Rozsahem viditelnosti proměnných definovaných v PHP je funkce. To znamená, že když někde uvnitř funkce definujeme proměnnou, je tato proměnná vidět v celém zbytku funkce (pokud ji neodnastavíme).

```
<?php
function variableScope() {
    for ($i=0; $i < 1; $i++) {
        $a = 0;
    }
    // proměnná $a je viditelná až do konce funkce - tedy i zde
}

variableScope();
// zde už proměnná $a viditelná není
?>
```

Funkce standardně nevidí globální proměnné s výjimkou tzv. superglobálních proměnných. To jsou speciální proměnné jako *$_GET*, *$_POST* a další, které PHP definuje pro data získaná zvenku.

199 Přístup ke globálním proměnným z funkce

Pro přístup ke globálním proměnným z funkce se dá použít buď pole *$GLOBALS*, které zpřístupňuje všechny globální proměnné, nebo deklarace klíčovým slovem `global`.

```
<?php
$user = array("login" => "test");

function printUserLogin() {
    global $user;
    echo "$user[login]\n";
    echo $GLOBALS["user"]["login"] . "\n";
}
?>
```

Ve většině případů je lepší se použití globálních proměnných z funkcí vyhnout, protože funkce by měla mít jasně definovaný vstup určený jejími parametry a výstup pouze v návratové hodnotě, případně parametrech předaných referencí. Když už globální proměnné použijeme, tak bychom do nich z funkcí alespoň neměli zapisovat.

200 Reference a globální proměnné

Při použití globálních proměnných spolu s referencemi je vhodné přesně chápat význam klíčového slova `global`. To ve skutečnosti definuje lokální proměnnou, která je referencí na globální proměnnou.

```
<?php
function referenceGlobalB(&$a) {
    global $b; // vyhodnotí se jako $b = &$GLOBALS["b"];
    $b = &$a; // změní referenci lokálního $b, to se navenek neprojeví
}
$b = null;
$a = 5;
referenceGlobalB($a);
// $b zůstane nastavené na null
?>
```

Poznámka: Pokus o provádění takovýchto „vylomenin“ obvykle svědčí o nevhodném návrhu aplikace a je lepší se nad ním znovu zamyslet.

201 Proměnné proměnné

PHP dovoluje pro přístup k proměnné použít syntaxi `$$var`. Při této syntaxi se přistoupí k proměnné, která se jmenuje stejně jako hodnota proměnné `$var`:

```
<?php
$var = "hello";
echo $$var; // vypsání proměnné $hello
?>
```

Obdobný trik lze použít i pro funkce: `$name()` zavolá funkci s názvem uloženým v proměnné `$name`. V naprosté většině případů pro tuto syntaxi ale není rozumné využití.

Poznámka: Proměnné proměnné nelze použít pro přístup ke speciální proměnné `$this`.

202 Proměnné s nepovolenými znaky

Ve skriptu můžou např. načtením z databáze vzniknout proměnné se znaky nepovolenými v identifikátorech (např. začínající číslem). K těmto proměnným můžeme přistoupit buď pomocí proměnných proměnných, nebo pomocí speciální syntaxe:

```
<?php
// proměnné proměnné
$var = "2persons";
$$var = 5;

// speciální syntaxe
echo ${"2persons"};
?>
```

203 Proměnné s rozšířenými znaky

PHP dovoluje v identifikátorech používat i znaky z horní poloviny ASCII tabulky. To může vést k některým těžko odhalitelným chybám:

```
<?php
$a = true;
// před rovnítkem je pevná mezera, vznikne tedy proměnná '$a '
?>
```

S odhalením takovýchto chyb nám může pomoci chybová úroveň `E_NOTICE` nebo zobrazení bílých znaků v editoru. Chyba může vzniknout např. zkopírováním textu z textového procesoru nebo z nevhodně formátovaného výpisu na webu.

204 Je proměnná nastavena?

Pokud se pokusíme pracovat s nedefinovanou proměnnou, vyvolá PHP chybu úrovně `E_NOTICE`. Proměnnou můžeme otestovat konstrukcí `isset`. Konstrukce vrátí `false` nejen na nedefinované proměnné, ale i na proměnné obsahující hodnotu `null`.

```
<?php
$null = null;
$defined = true;
var_dump(isset($notDefined)); // false
var_dump(isset($null)); // false
var_dump(isset($defined)); // true
?>
```

Poznámka: Pokud chceme zjistit, jestli existuje prvek s nějakým indexem v poli (i když by měl hodnotu `null`), můžeme k tomu použít funkci `array_key_exists`. U globálních proměnných můžeme použít tuto funkci s polem *`$GLOBALS`*, u lokálních proměnných můžeme použít výsledek funkce `get_defined_vars`. Ve většině případů lze ale s hodnotou `null` zacházet stejně jako s nedefinovanou proměnnou, takže stačí použít konstrukci `isset`.

205 Jsou proměnné nastavené?

Konstrukcí `isset` můžeme otestovat i více proměnných najednou:

```
<?php
$defined = true;
var_dump(isset($defined, $notDefined));
?>
```

Konstrukce vrátí `true` jen v případě, že jsou nastavené všechny parametry.

206 Zrušení proměnné


pokročilý

Pokud chceme proměnnou uvolnit z paměti, můžeme použít jazykovou konstrukci `unset`. Lze ji zavolat i s více parametry.

```
<?php
$defined = true;
unset($defined);
?>
```


Poznámka: Na konci skriptu se o uvolnění všech proměnných stará PHP, takže u většiny skriptů se o něj explicitně starat nemusíme. Hodí se to obvykle jen u dlouho běžících skriptů.

207 Přiřazení proměnných hodnotou


začátečník

Operátor přiřazení v PHP vytváří nezávislou kopii, takže proměnná a přiřazovaná hodnota jsou na sobě nezávislé. Neplatí to pouze u typu `resource` a od PHP 5 také u objektů, kde přiřazení jedné proměnné do druhé způsobí, že pracují se stejným objektem.

```
<?php
$array = array(1);
$copy = $array; // nezávislá kopie
$copy[0]++;
echo $array[0]; // vypíše 1

$obj = new stdClass;
$obj->x = 1;
$copy = $obj; // obě proměnné pracují se stejným objektem
$copy->x++;
echo $obj->x; // vypíše 2
?>
```


Poznámka: Stejně se chová i předávání parametrů funkcím.

208 Přiřazení proměnných referencí


pokročilý

Pokud chceme, aby proměnná ukazovala na stejné místo v paměti jako nějaká jiná proměnná, tak ji můžeme přiřadit referencí. Jde tedy vlastně o jakýsi alias. Tento obrat se používá hlavně tehdy, když chceme opakovaně pracovat s prvkem uvnitř víceúrovňového pole. K přiřazení proměnné referencí se používá operátor `&`.

```
<?php
$array = array(array(1));
$ref = &$array[0][0];
$ref++; // změní se i hodnota uvnitř pole
echo $array[0][0]; // vypíše 2
?>
```


Poznámka: Vazba mezi proměnnými zanikne, když jednu z nich odnastavíme konstrukcí `unset`.

209 Předání parametrů referencí

Uživatelské funkce standardně dostávají argumenty předané hodnotou. To znamená, že změny, které v argumentech provedou, se neprojeví navenek funkce. Pokud se změny navenek projevit mají, je nutné parametry předávat referencí. To zajistíme znakem `&` uvedeným před názvem parametru v deklaraci funkce.

```
<?php
function increment(&$param) {
    $param++;
}
$a = 1;
increment($a);
echo $a; // vypíše 2
?>
```

Poznámka: Předávání parametrů referencí je obvykle lepší se vyhnout. Funkce v PHP můžou vracet libovolně složité hodnoty a většinou není důvod ukládat výsledek do předaných argumentů.

210 Předání parametrů referencí při volání

Parametry předávané referencí lze určit až při volání funkce. Tomuto způsobu je ale lepší se vyhnout, protože funkce může parametry modifikovat způsobem, se kterým nepočítáme.

```
<?php
function getIncrement($param) {
    $param++;
    return $param;
}
$a = 1;
getIncrement(&$a);
echo "$a\n"; // vypíše 2
?>
```

Poznámka: Toto chování je označeno jako zastaralé a lze ho zakázat konfigurační direktivou `allow_call_time_pass_reference`.

211 Přiřazení objektů referencí

Často se zjednodušeně uvádí, že objekty se od PHP 5 přiřazují vždy referencí. To ale není úplně přesné – pokud do jedné z proměnných přiřadíme novou hodnotu, tak to druhou proměnnou na rozdíl od přiřazení referencí nezmění:

```
<?php
$obj = new stdClass;
$obj->x = 1;
$copy = $obj;
$copy = "copy"; // v proměnné $obj zůstane původní objekt
```

```
$ref = &$obj;
$ref = "ref"; // v proměnné $obj bude nyní "ref"
?>
```

212 Volitelný parametr předávaný referencí

Od PHP 5 mohou mít i parametry předávané referencí výchozí hodnotu, takže mohou být nepovinné. Tento obrat využívají některé funkce PHP, které do nepovinného parametru předávaného referencí zapisují chybovou hlášku (např. `fSockOpen`). Stejného chování můžeme tedy dosáhnout i v uživatelské funkci.

213 Jak použít přiřazení jako výraz

V kódu se dá využít toho, že přiřazení je v jazyce PHP výraz. Kromě přiřazení hodnoty do proměnné také vrátí přiřazovanou hodnotu, která se dá dále využít:

```
<?php
// inicializace více proměnných najednou
$a = $b = array();

// přiřazení a zároveň otestování pravdivosti
while ($row = mysql_fetch_assoc($result)) {
    // kód se provádí, dokud mysql_fetch_assoc() nevrátí false
    // jeho výsledek se zároveň přiřadí do proměnné $row
}
?>
```

Kvůli čitelnosti však není vhodné tento obrat používat příliš často. První ukázka nám může ztížit hledání inicializace proměnné `$b`, druhá ukázka může v začátečníkovi zase vzbudit dojem porovnávání.

214 Zachování typu proměnné

PHP je dynamicky typovaný jazyk, což znamená, že proměnná je toho typu, jaká je v ní přiřazená hodnota. Toho se dá někdy obratně využít, obvykle je ale lepší v proměnné udržovat stále stejný typ. Jedna z výhod je, že při ladění pak podle obsahu proměnné snadno poznáme, jaký datový typ by měla obsahovat.

```
<?php
// vhodná inicializace prázdného pole
$array = array();

// méně vhodná inicializace
$array = null;
?>
```

Stejná zásada platí i u nepovinných parametrů funkcí.

215 Zachování významu proměnné

Do stejné proměnné není vhodné ukládat hodnoty s různým významem, i kdyby byly stejného datového typu. Ztěžuje to další rozšiřování programu a zhoršuje možnosti ladění.

```
<?php
// v proměnné zůstane řetězec, ale s jiným významem
$user = "Jméno";
$user = mysql_result(mysql_query("
    SELECT login
    FROM user
    WHERE name = '" . mysql_real_escape_string($user) . "'
"), 0);
// vhodnější je definovat proměnné $userName a $userLogin
?>
```

216 Zjištění typu proměnné

Pro zjištění typu proměnné můžeme použít funkci `getType`. Na základě toho bychom mohli třeba upravit chování funkce, to ale není příliš vhodná praktika. Mnohem častěji chceme ověřit, zda je proměnná nějakého konkrétního typu, a pokud ne, tak třeba vyvolat chybu. To zajišťují funkce jako `is_int`, `is_array`, `is_object` a podobné.

217 Ověření typu řetězec

K ověření typu řetězec se dá použít funkce `is_string`. V naprosté většině kontextů se ale jako s řetězcem dá pracovat i s ostatním skalárními hodnotami (např. čísly) a jako řetězce se můžou chovat i objekty implementující metodu `__toString`.

```
<?php
/** Ověření, zda se s proměnnou dá pracovat jako s řetězcem
* @param mixed
* @return bool
*/
function isLikeString($var) {
    return !isset($var) || is_scalar($var)
        || method_exists($var, '__toString')
    ;
}
?>
```

218 Nastavení typu proměnné

Pro změnu typu proměnné se dá použít funkce `setType`. Tu bychom použili v případě, když bychom výsledný typ měli uložený např. v proměnné. Obvykle ho ale známe, takže můžeme použít přetypování:

```
<?php
$id = (int) $_GET["id"];
?>
```

Poznámka: Datový typ můžeme uvádět ve zkráceném i plném zápisu, tedy např. `(int)` i `(integer)`.

219 Převod na pravdivostní hodnotu

Operátor `!` vytvoří převrácenou pravdivostní hodnotu výrazu. Pokud tedy chceme nějaký výraz převést na `true` nebo `false`, můžeme použít `!!výraz`. Tento zápis je sice krátký, ale značně nepřehledný, proto je lepší používat explicitní přetypování ve tvaru `(bool) výraz`.

220 Jaké používá PHP velikosti čísel

Celá čísla se v PHP ukládají vždy se znaménkem ve velikosti závislé na platformě (obvykle 32 nebo 64 bitů). Větší čísla se automaticky převádí na desetinná, ta se ale neukládají přesně (přesnost je zhruba 14 číslic). Tomuto faktu je vhodné přizpůsobit např. i strukturu databáze.

221 Jak pracovat s přesnými čísly

Pro přesnou práci s velkými nebo desetinnými čísly lze použít knihovnu **BCMath**:

```
<?php
var_dump(bcAdd('1234567890', '9876543210'));
// velká čísla se zadávají jako řetězce
?>
```

Poznámka: Méně používanou alternativu představuje knihovna GMP.

222 Bitový posun

Operátory bitového posunu můžeme použít např. při definici konstant vyjadřujících nějakou masku.

```
<?php
define("SEARCH_HEADING", 1 << 0);
define("SEARCH_PEREX", 1 << 1);
define("SEARCH_BODY", 1 << 2);
define("SEARCH_ALL", (1 << 3) - 1);

if ($search & SEARCH_PEREX) {
    // vyhledání v perexu
}
?>
```

Tento způsob zápisu jednotlivých bitů je názornější než uvádění desítkových čísel, ve kterých se hlavně u vyšších hodnot snadno udělá chyba. Nejde bohužel použít při definici konstant pomocí klíčového slova `const`.

223 Zápis hexadecimálních čísel

Pro zápis hexadecimálních čísel lze použít notaci `0xA0`. Kromě toho existuje funkce `hexDec` a inverzní `decHex` pro převod mezi šestnáctkovou a desítkovou soustavou.

Hexadecimální čísla se dají použít pro celkem přehledný zápis bitových hodnot. Binární čísla totiž vlastní notaci nemají (dají se nicméně zpracovat funkcí `binDec`).

224 Zápis čísel v osmičkové soustavě

Pro zápis čísel v osmičkové soustavě se dá použít notace `0755` nebo funkce `octDec`, která převede řetězec z osmičkové soustavy.

Osmičková čísla nachází využití hlavně u funkce `chMod` případně `mkDir`.

225 Zápis čísel ve vědecké notaci

PHP dovoluje zapisovat čísla v tzv. vědecké notaci, tedy např. jako `1e6` ($1 * 10^6$). Hodí se to hlavně u velkých čísel, kdy je tento zápis přehlednější než jednička následovaná mnoha nulami. Tímto způsobem vznikne vždy desetinné číslo (typ `float`).

226 Převod uživatelského vstupu na číslo

Často potřebujeme převést uživatelský vstup na číslo. Pokud bychom např. důvěřovali identifikátoru článku předanému v URL, může to vést k chybě nebo bezpečnostnímu problému. PHP nabízí hned několik způsobů převodu řetězce na číslo:

- Přetypování pomocí konstrukce `(int) $_GET["id"]`.
- Použití funkce `intVal($_GET["id"])`.
- Použití matematické operace, např. `1 * $_GET["id"]` nebo nejsnáze pomocí unárního plus: `+$_GET["id"]`. To může ale vytvořit i desetinné číslo.

Za nejčitelnější považujeme použití funkce, která se dá navíc hromadně aplikovat i na prvky pole:

```
<?php
$ids = array_map('intVal', $_GET["ids"]);
?>
```

227 Kontrola číselného řetězce

Pokud má uživatel např. do formuláře vyplnit nezáporné celé číslo a my potřebujeme zjistit, jestli nevyplnil něco jiného, můžeme použít funkci `ctype_digit`. Funkce `is_numeric` se k tomuto účelu nehodí, protože přijme všechny formáty, které se v PHP dají pro zápis čísla použít (tedy např. i `-1.2e+5` nebo `0xFF`).

```
<?php
if (!ctype_digit($_POST["number"])) {
```

```
    echo "Vyplňte prosím číslo.\n";
}
?>
```

228 Používání čísel v kódu


pokročilý

Někdy v kódu potřebujeme použít nějaké číslo. Např. chceme nastavit nějakou hodnotu zadávanou v sekundách na jeden den. Máme v zásadě tři možnosti, jak to udělat:

1. Číslo do kódu přímo napsat (`86400`). Pak je ale vhodné alespoň do komentáře poznamenat, co vlastně znamená.
2. Do kódu umístit výpočet, kterým lze číslo odvodit (`24 * 60 * 60`). Z toho je lépe patrné, co číslo znamená.
3. Vytvořit konstantu a používat ji místo čísla (`DAY_IN_SEC`). Tento způsob můžeme zkombinovat s předchozím.

```
<?php
define("DAY_IN_SEC", 24 * 60 * 60); // počet sekund za den
echo 2 * DAY_IN_SEC; // dva dny v sekundách
?>
```


Poznámka: Pokud hodnotu používáme na více místech (např. výše DPH), tak bychom ji vždy měli nastavit centrálně, např. v konstantě.

229 Nastavení jazyka


začátečník

Řada funkcí PHP respektuje jazyk a kódování nastavené funkcí `setLocale`. Týká se to řetězcových funkcí (např. `strToUpper`), porovnávání řetězců funkcí `strColl`, formátování desetinných čísel funkcí `sPrintF` nebo formátování data funkcí `strFTime`.

Problém funkce `setLocale` je v tom, že různé platformy podporují různé jazyky a kódování, takže se na jednotné nastavení nedá spolehnout. Pokud má tedy aplikace běžet v různých prostředích, je lepší místo tohoto konceptu používat např. extenze **MBstring** a **Intl**, které se chovají všude stejně:

```
<?php
// jazyk C znamená výchozí jednotné nastavení
setLocale(LC_ALL, "C");
?>
```

230 Řetězec jako posloupnost bajtů


pokročilý

Řetězec je v PHP tvořen posloupností bajtů, nikoliv znaků. To je důležité vědět při používání vícebajtových kódování, např. UTF-8. Pro práci s nimi lze používat funkce knihovny **MBstring** nebo **iconv**.

231 Přístup k jednomu znaku v řetězci


pokročilý

Pro přístup ke konkrétnímu znaku (přesněji řečeno bajtu) v řetězci můžeme použít hranaté závorky s číselným indexem počítaným od nuly.

```
<?php
echo $s[0]; // vypíše první bajt řetězce
?>
```


Poznámka: Místo hranatých závorek lze použít i složené (stejně jako pro přístup k prvku pole), tato syntaxe ale byla označena za zastaralou.

232 Binární bezpečnost řetězcových funkcí


pokročilý

PHP je vytvořeno v programovacím jazyce C, kde se řetězce ukončují nulovým bajtem. V prehistorických verzích PHP byl problém s binárními řetězci, které obsahovaly tento bajt, protože je některé vestavěné funkce ořízly. Současné verze ale už tímto problémem netrpí, a jsou tedy binárně bezpečné – dokážou pracovat i s binárními daty.

233 Speciální znaky v řetězci


začátečník

V uvozovkovém řetězci lze za zpětným lomítkem uvádět celou řadu znaků se zvláštním významem. Nejčastěji se používá `\n` (konec řádku), `\r\n` (konec řádku používaný např. v hlavičkách e-mailu) a `\t` (tabulátor). Znaky `\v` a `\f` se příliš nepoužívají, užitečná je ale sekvence `\x` následovaná dvěma hexadecimálními číslicemi, která se dá použít pro zápis libovolného binárního znaku. Ty lze sice zapsat i přímo, ale znaky v jiném kódování, než které má skript, by mohly rozhodit některé editory.

```
<?php
/** Převod kódování CP1250 na Latin2
* @param string řetězec v kódování CP1250
* @return string řetězec v kódování Latin2
*/
function cp1250ToLatin2($string) {
    return strTr($string,
        "\x8A\x8D\x8E\x9A\x9D\x9E",
        "\xA9\xAB\xAE\xB9\xBB\xBE"
    );
}
?>
```

234 Znaková sada a kódování


začátečník

Znaková sada je soubor znaků, které lze nějakým způsobem vyjádřit. Kódování je způsob vyjádření těchto znaků.

Například znaková sada Unicode obsahuje prakticky všechny znaky, které se na světě používají. Kódování UTF-8 nebo UTF-16 jsou potom zápisem této znakové sady. Obě kódování tedy dovolují zapsat stejné znaky, ale používají k tomu jiný bajtový zápis.

Oba pojmy se často zaměňují a pořádek v nich nemají ani autoři specifikací. Např. parametr `charset` (z anglického *character set*, znaková sada) hlavičky `Content-Type` ve skutečnosti definuje kódování.

235 Jak na převod kódování

Pro převod kódování lze v PHP použít funkci `iconv`. V extenzi **MBstring** existuje i funkce `mb_convert_encoding`, ta ale nepodporuje běžná kódování používaná pro češtinu.

Pro jednoduchý převod kódování ISO-8859-1 na UTF-8 lze použít také funkci `utf8_encode`, obrácený převod zajišťuje funkce `utf8_decode`. Nepodporované znaky (např. ty české) jsou převedeny na otazník.

236 Délka řetězce v kódování UTF-8

Pro zjištění délky řetězce v kódování UTF-8 se dají použít funkce `iconv_strLen` nebo `mb_strLen`, ty ale vyžadují povolení patřičných extenzí. Řešení nezávislé na extenzi je převést řetězec funkcí `utf8_decode` a jeho délku zjistit běžnou funkcí `strLen`. Nepodporované znaky se sice převedou na otazníky, to ale pro samotné zjištění délky nevadí.

```
<?php
if (strLen(utf8_decode($_POST["name"])) > 20) {
    echo "Maximální délka jména je 20 znaků.\n";
}
?>
```

Poznámka: Extenze iconv je běžně povolena.

237 Samotné zpětné lomítko v řetězci

Pokud v řetězci uvedeme zpětné lomítko, za kterým nenásleduje znak se speciálním významem, stane se zpětné lomítko součástí řetězce. Uvnitř uvozovek není vhodné se na to spoléhat, protože v budoucích verzích PHP můžou přibýt další znaky se speciálním významem. Uvnitř apostrofů toho využít lze.

```
<?php
// všechny řetězce budou obsahovat dva znaky
$a = "\a"; // riskantní kvůli budoucím verzím PHP
$b = "\\a"; // v pořádku
$c = '\a'; // v pořádku
$d = '\\a'; // pečlivé
?>
```

238 Jak na víceřádkové řetězce

PHP dovoluje uvnitř řetězců, na rozdíl od většiny ostatních programovacích jazyků, používat znak konce řádku jeho prostým uvedením:

```
<?php
echo "Řetězec
přes
čtyři
řádky";
?>
```

V řetězcích tak není nutné používat symbol `\n`, i když se to z prostorových důvodů často stejně dělá. Stejně tak není nutné pro vytvoření víceřádkového řetězce používat tzv. *heredoc* syntaxi.

Poznámka: Víceřádkový zápis řetězců není vhodné používat na místech, kde záleží na použitém oddělovači řádků – např. při komunikaci HTTP protokolem je nutné řádky ukončovat pomocí `\r\n`, a nikoliv samotným `\n`. Některé systémy pro správu verzí totiž mohou znak konce řádku změnit.

239 Co je to heredoc

PHP pro zápis víceřádkových řetězců dovoluje používat tzv. *heredoc* syntaxi. Její výhoda proti běžnému uvozovkovému řetězci spočívá v tom, že uvozovka nemá zvláštní význam, a může se tedy zapsat bez ošetření. Proměnné a speciální znaky se zpracovávají stejně jako uvnitř uvozovek.

```
<?php
// od PHP 5.3 lze použít také <<<"EOT"
echo <<<EOT
víceřádkový text
včetně "uvozovek"
EOT;
?>
```

Poznámka: Heredoc syntaxe se před PHP 5.3 nedá použít pro inicializaci vlastností objektů.

240 Řetězec bez interpretace speciálních znaků

Někdy chceme zapsat řetězec s řadou speciálních znaků (uvozovky, apostrofy, zpětná lomítka a dolary). Od PHP 5.3 k tomu můžeme použít tzv. *nowdoc* syntaxi, kde se žádné speciální znaky neinterpretují.

```
<?php
echo <<<'EOT'
Libovolný řetězec včetně 'apostrofů', "uvozovek",
\zpětných lomítek a $dolarů.
EOT;
?>
```

241 Přístup k prvkům pole uvnitř řetězce

Pokud přistupujeme k řetězcovému klíči pole, měli bychom ho vždy uzavřít do apostrofů nebo uvozovek. Zápis `$row[id]` se totiž pokusí vyhledat konstantu `id`, a teprve když ji nenajde, vygeneruje chybu a místo ní použije řetězec. Pokud bychom tedy v budoucnu konstantu daného názvu definovali, přestane kód fungovat.

Uvnitř řetězce uzavřeného do uvozovek je ale situace jiná. Zápis `"$row[id]"` hledá vždy řetězcový klíč `id`, protože už jsme v kontextu řetězce, kde se konstanty nevyhodnocují.

```
<?php
$row[id]; // vyvolá chybu a přestane fungovat po definici konstanty id
$row["id"]; // v pořádku
"$row[id]"; // v pořádku
"$row['id']"; // syntaktická chyba
?>
```

242 Přístup ke složitějším strukturám z řetězce

Z řetězce lze přímo přistupovat ke skalárním proměnným, prvkům pole s klíčem ve tvaru identifikátoru a vlastnostem objektu. Následující zápisy jsou tedy povolené:

```
<?php
"$array[key]";
"$obj->key";
"$array[$key]";
?>
```

Složitější obraty (např. přístup k prvku vícerozměrného pole) povolené nejsou a je nutné je zapsat mimo řetězec: `"Hodnota: " . $array["key"]["key2"] . "\n"` nebo pomocí složených závorek.

243 Přístup k libovolným strukturám z řetězce

Pomocí složených závorek lze z řetězce přistoupit i k libovolným složitějším strukturám:

```
<?php
echo "Hodnota: {$array["key"]["s"]}\n";
?>
```

Poznámka: Pomocí této syntaxe lze dokonce volat např. i metody objektu, to ale kód spíš znepřehlední.

244 Je řetězec neprázdný?

Pokud chceme zjistit, jestli je řetězec neprázdný, nemůžeme použít automatické přetypování na pravdivostní hodnotu, protože řetězec `"0"` se v PHP vyhodnotí jako nepravda:

```
<?php
if ($s) {
    // nespustí se pro $s == "0"
}
?>
```

Pokud může výraz obsahovat výhradně řetězec, můžeme použít explicitní porovnání `$s != ""`. Pokud by ale proměnná mohla obsahovat i číslo, tento zápis opět nebude spolehlivý:

```
<?php
if ($x != "") {
    // nespustí se pro $x == 0
}
?>
```

Spolehlivý způsob zjištění, jestli se výraz převede na neprázdný řetězec, nabízí funkce `strLen`.

```
<?php
if (!strLen($x)) {
    echo "(prázdný řetězec)";
}
?>
```

Použít lze také přetypování na řetězec a použití operátoru identity: `(string) $x === ""` nebo porovnání s hodnotou `null`: `$x == null`.

245 Porovnávání řetězců

Pro porovnání hodnoty řetězců lze použít běžné operátory jako < nebo >=. Tyto operátory porovnávají stejně jako funkce `strCmp` binární hodnotu jednotlivých bajtů řetězců. Důsledkem toho je, že rozlišují velikost písmen. Pokud chceme provést porovnání se zohledněním *locales*, můžeme použít funkci `strColl`. Když chceme porovnat dva řetězce ve vícebajtovém kódování, můžeme použít metodu `Collator::compare`.

246 Porovnání řetězců bez rozlišení velikosti písmen

Pro porovnání řetězců bez rozlišení velikosti písmen se dají oba převést na stejnou velikost funkcí `strToLower` nebo `strToUpper`. Efektivnější je nicméně použít funkci `strCaseCmp`, která je k tomu přímo určená.

```
<?php
if (strToLower($name) == "define") {
}

// efektivnější způsob
if (strCaseCmp($name, "define") == 0) {
}
?>
```

247 Porovnání číselných řetězců

PHP jako dynamicky typovaný jazyk obsahuje pravidlo pro porovnání dvou řetězců obsahujících čísla. Ty se před porovnáním převedou na čísla a porovnají se tato čísla. Důsledkem toho je, že `"1" == "01"`, `"0" == "00"` a tak dále. Pokud nám toto pravidlo u některých řetězců nevyhovuje, je lepší použít operátor neidentity, který typy nepřevádí: `"1" !== "01"`.

248 Porovnání prvních několika znaků řetězce

Pro porovnání několika prvních znaků řetězce se dá použít funkce `strNCmp`. Často je potřeba zjistit, jestli řetězec něčím začíná. Za tím účelem si můžeme vytvořit vlastní funkci:

```
<?php
/** Zjištění, zda má řetězec daný začátek
* @param string zkoumaný řetězec
* @param string požadovaný začátek
* @return bool
*/
function stringStartsWith($string, $start) {
    return (strNCmp($string, $start, strLen($start)) == 0);
}
?>
```

249 Končí text tímto řetězcem?

Občas je potřeba zjistit, jestli řetězec něčím končí. Můžeme k tomu využít záporný index funkce `substr`:

```
<?php
/** Zjištění, zda má řetězec daný začátek
* @param string zkoumaný řetězec
* @param string požadovaný konec
* @return bool
*/
function stringEndsWith($string, $end) {
    if (strLen($end) == 0) {
        return false;
    }
    return (substr($string, -strLen($end)) == $end);
}
?>
```

250 Záměna více řetězců najednou

Někdy se může stát, že v řetězci chceme najít a nahradit více podřetězců najednou. To se dá udělat několika možnými způsoby:

```
<?php
// neefektivní
$s = str_replace(" ", " ", $s);
$s = str_replace("&shy;", "", $s);

// u dlouhých řetězců neefektivní - řetězec se stále prochází dvakrát
$s = str_replace(array(" ", "&shy;"), array(" ", ""), $s);

// řetězec se prochází jen jednou, zároveň je nejčitelnější
$s = strTr($s, array(" " => " ", "&shy;" => ""));
?>
```

251 Záměna více bajtů najednou

Pokud v řetězci chceme najednou nahradit více bajtů, můžeme k tomu použít funkci `strTr` se třemi parametry.

```
<?php
// převod kódování CP1250 na Latin2
strTr($s, "\x8A\x8D\x8E\x9A\x9D\x9E", "\xA9\xAB\xAE\xB9\xBB\xBE");
?>
```

Funkce `strTr` chápe řetězce jako posloupnost bajtů, nemůžeme ji v této podobě tedy použít např. pro nalezení znaků v kódování UTF-8.

252 Ošetřování parametrů funkcí

Pokud si funkce potřebuje data nějakým způsobem ošetřit (třeba proto, aby je mohla vypsat do HTML stránky nebo poslat do databáze), měla by přijímat čistá data a ošetřit si je sama.

```
<?php
// správný návrh
function htmlHeader($title) {
    echo "<title>" . htmlSpecialChars($title) . "</title>\n";
}
htmlHeader("Já & syn");

// nesprávný návrh
function htmlHeaderEscaped($title) {
    echo "<title>$title</title>\n";
}
htmlHeaderEscaped("Já &amp; syn");
?>
```

253 Ošetření chyby při deserializaci proměnných

Pokud ve funkci `unserialize` dojde k chybě, tak tato funkce vrátí stejně jako většina ostatních funkcí PHP hodnotu `false`. Tuto hodnotu ovšem vrátí samozřejmě i při deserializaci hodnoty `false`. Abychom tuto nejednoznačnost vyřešili, můžeme serializovanou hodnotu porovnat se serializací hodnoty `false`:

```
<?php
$original = unserialize($serialized);
if ($original === false && $serialized !== serialize(false)) {
    // tady bude ošetření chyby
}
?>
```