User Defined Functions, Namespaces

Disclaimer: вы смотрите просто запись лекции, это **НЕ** специально подготовленный видеокурс!

Some important facts about functions in PHP

A function may accept parameters passed by value or by reference.

A function may accept variable parameters count.

A function parameters may have default values.

Parameters order may differ in function definition and function call.

Defining a function: general approach

The simplest approach to function definition and usage looks like this:

```
<?php

function sqr($x)
{
    return $x * $x;
}

$y = sqr(2);
echo $y; // 4
```

Since PHP 8 you may put a comma after the last argument, i.e.:

function myFnc($a, $b, $c,)

Defining a function: general approach

And now – with types:

```php
<?php

function sqr(float|int $x) : float|int
{
    return $x * $x;
}

$y = sqr(2);
echo $y; // 4
```

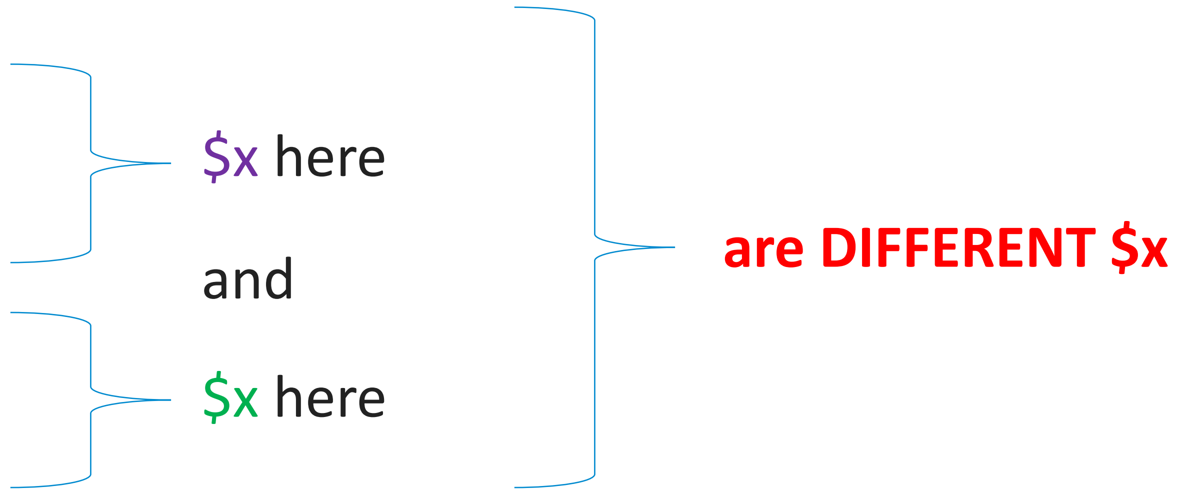
Variable scope

Variables from outside a function body are not accessible within a function (and vice versa). This is also applicable to passed-by-value parameters.

```php
<?php

function fnc($x)
{
    $x = 99;
}

$x = 55;
fnc($x);
echo $x; // 55
```



Variable scope

Q: Is there a way to access any global variable from within a function body?
A: Yes (‘global’ keyword and $GLOBALS array), but DON’T DO THIS! This is a terrible anti-pattern!

Passed-by-value parameters

Passed-by-value parameters produce local copies. Any changes made to that copies do not affect the original variable.

```php
<?php

function fnc($x)
{
    $x = $x + 10;
    return $x;
}

$x = 10;
$y = fnc($x);
echo 'X = ' . $x . '; Y = ' . $y; // X = 10; Y = 20
```

Objects are always passed-by-reference!

Passed-by-reference parameters

Passed-by-reference parameters produce ‘links’ to the same memory address. So, any changes here will affect the ‘original variable’.

```php
<?php

function fnc(&$y)
{
    $y = $y + 10;
}

$x = 10;
fnc($x);
echo 'X = ' . $x; // X = 20
```

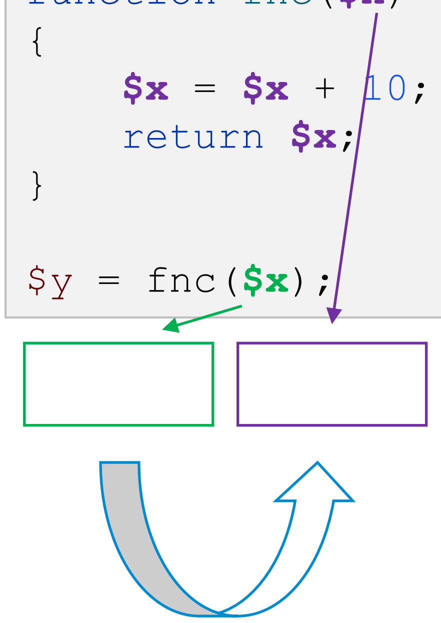
Objects are always passed-by-reference! Even without &!

Passed-by-value and passed-by-reference parameters

Passed-by-value

```
function fnc($x)
{
     $x = $x + 10;
     return $x;
}

$y = fnc($x);
```

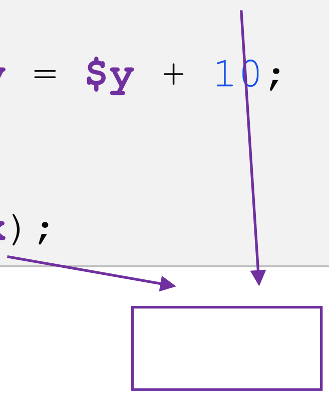


A value is copied to another memory area.

Passed-by-reference

```
function fnc(&$y)
{
     $y = $y + 10;
}

fnc($x);
```



Both variables are linked to the same memory area.

Default parameter values

One may pre-define a value for a parameter (this value is used when actual value was not passed). Such parameters with default values should be placed to the right from ‘ordinary parameters’.

```php
<?php

function fnc($x, $y = 99)
{
    echo 'X = ' . $x . ';  Y = ' . $y; // X = 10; Y = 99
}

fnc(10);
```

Differentiating parameters order

Since PHP 8 it is possible to use parameters names during functions call. This may look rather strange, but it allows the following:

```php
<?php

function test(string $a, string $b, ?string $c = null, ?string $d = null)
{
    var_dump(func_get_args());
}

test(
b: 'value b',
a: 'value a',
d: 'value d',
);
```

```
/*
array(4) {
  [0] =>
  string(7) "value a"
  [1] =>
  string(7) "value b"
  [2] =>
  NULL
  [3] =>
  string(7) "value d"
}
*/
```

Variable parameters count

In many cases it is convenient to pass as many parameters as you like, there are special service function to deal with this situation:

```php
<?php

function fnc()
{
    echo func_num_args();     // 3
    echo func_get_arg(1);     // B
    print_r(func_get_args()); // Array([0] => A [1] => B [2] => C)
}

fnc('A', 'B', 'C');
```

Variable parameters count

And there is another approach – a special syntax to deal with variable parameters count:

```php
<?php

function fnc($a, $b, ...$c)
{
    echo $a;      // A
    echo $b;      // B
    print_r($c); // Array ([0] => C [1] => D [2] => E)
}

fnc('A', 'B', 'C', 'D', 'E');
```

Static variables

Static variables are initialized during the first function call. Then such variables preserve their values between further function calls.

```php
<?php

function fnc()
{
    static $staticCounter = 1;
    $normalCounter = 1;

    echo 'Static counter: ' . $staticCounter . ', Normal counter: ' . $normalCounter . "\n";

    $staticCounter++;
    $normalCounter++;
}

fnc(); // Static counter: 1, Normal counter: 1
fnc(); // Static counter: 2, Normal counter: 1
fnc(); // Static counter: 3, Normal counter: 1
```

Anonymous functions and closures

Anonymous functions, also known as closures, allow the creation of functions which have no specified name. They are most useful as the value of callable parameters, but they have many other uses.

Anonymous functions and closures

The most common case of anonymous functions usage is a situation when we need a function for just one single call, let’s see an example:

```php
<?php

// Let's assume we want to order these words by length
$words = array('ABC', 'A', 'LongWord');

// Classic approach forces us to define a function (comparator)
function compareByLength(string $a, string $b) : int
{
    return (strlen($a) <=> strlen($b));
}

usort($words, 'compareByLength');

// Or we may just use an anonymous function
usort($words, function ($a, $b) {
    return (strlen($a) <=> strlen($b));
});

print_r($words); //Array ([0] => A [1] => ABC [2] => LongWord)
```

Anonymous functions and closures

Closures ‘remember’ some variables used during closure definition. This allows us to create a kind of ‘function factory’:

```php
<?php

$words = array('ABC', 'A', 'LongWord');

// The simplest closure
$reverseLengthComparator = function () {
    return function ($a, $b) {
        return -(strlen($a) <=> strlen($b));
    }; // Yes, here we need ;
}; // Yes, here we need ;

usort($words, $reverseLengthComparator());
print_r($words);
```

```php
// And here we have a real function factory
$comparatorsFactory = function ($method) {
    return function ($a, $b) use ($method) {
        if ($method == 'alphabet') {
            return ($a <=> $b);
        }
        if ($method == 'codesums') {
            $cs_a = 0;
            $cs_b = 0;
            for ($i = 0; $i < strlen($a); $i++) {
                $cs_a += ord($a[$i]);
            }
            for ($i = 0; $i < strlen($b); $i++) {
                $cs_b += ord($b[$i]);
            }
            return ($cs_a <=> $cs_b);
        }
        return 0;
    };
};

// We may call our factory ad-hoc
usort($words, $comparatorsFactory('alphabet'));
print_r($words);

usort($words, $comparatorsFactory('codesums'));
print_r($words);

// Or we may produce comparators in advance and store
// them in some variables
$alphabetComparator = $comparatorsFactory('alphabet');
$codesumsComparator = $comparatorsFactory('codesums');

usort($words, $alphabetComparator);
print_r($words);

usort($words, $codesumsComparator);
print_r($words);
```

Arrow functions

Arrow functions were introduced in PHP 7.4 as a more concise syntax for anonymous functions.

```php
<?php

// Imagine we have an array of objects (some messages).
class Message
{
    private ?int $id = null;

    public function __construct(int $id)
    {
        $this->id = $id;
    }

    public function getId(): ?int
    {
        return $this->id;
    }
}

$messages = array();

for ($i = 0; $i < 10; $i++) {
    $messages[] = new Message($i);
}
```

```php
// And now we want to have an array of messages ids only.

// 1) Classic approach:

$idsOldWay = array();
foreach ($messages as $message) {
    $idsOldWay[] = $message->getId();
}

print_r($idsOldWay);

// 2) Closure approach:

$idsClosure = array_map(function ($message) {
    return $message->getId();
}, $messages);

print_r($idsClosure);

// 3) Arrow function approach:

$idsArrowFunction = array_map(fn ($message) => $message->getId(), $messages);

print_r($idsArrowFunction);
```

Generators

A generator function looks just like a normal function, except that instead of returning a value, a generator yields as many values as it needs to.

```php
<?php

function getSequence(int $start, int $end) {
    for ($i = $start; $i <= $end; $i++) {
        yield $i;
    }
}

$generator = getSequence(5, 10);
foreach ($generator as $value) {
    echo $value . " ";
}

// 5 6 7 8 9 10
```

Namespaces

In the PHP namespaces are designed to solve two problems that authors of libraries and applications encounter when creating re-usable code elements such as classes or functions:

- Name collisions between code you create, and internal PHP classes/functions/constants or third-party classes/functions/constants.
- Ability to alias (or shorten) Extra_Long_Names designed to alleviate the first problem, improving readability of source code.

Namespaces, simple usage sample

```php
<?php

namespace SomeProject\SomeLibrary\EasyURL;

class URL
{
    public $JustForTest_EesyURL = 1;
}
```

```php
<?php

namespace SomeProject\SomeLibrary\ComplexURL;

class URL
{
    public $JustForTest_ComplexURL = 1;
}
```

In both libraries there are URL classes (with the same name obviously).

```php
<?php
require_once('some_library_1.php');
require_once('some_library_2.php');

use \SomeProject\SomeLibrary\EasyURL as Easy;
use \SomeProject\SomeLibrary\ComplexURL as Complex;

$easyURL = new Easy\URL();
$complexURL = new Complex\URL();
var_dump($easyURL);
var_dump($complexURL);

/*
class SomeProject\SomeLibrary\EasyURL\URL#1 (1) {
  public $JustForTest_EesyURL =>
  int(1)
}
class SomeProject\SomeLibrary\ComplexURL\URL#2 (1) {
  public $JustForTest_ComplexURL =>
  int(1)
}
*/
```

Still there are no conflicts.

User Defined Functions, Namespaces

Disclaimer: вы смотрите просто запись лекции, это **НЕ** специально подготовленный видеокурс!