Language Fundamentals

Super Basics

Life in the PHP Interpreter

- Web server passes request to PHP when it encounters a file with a PHP file extension (almost always .php).
- PHP treats everything as HTML until it encounters a PHP open tag:
  - <?php “open tag” starts it processing as PHP
  - ?> “close tag” means stop parsing as PHP
  - Only use close tag if you have more HTML

Short Echo Tag

- <?= $variable ?> “short echo tag” displays (echoes) the contents of the variable or expression to the web server, then returns to HTML processing
- Example: I see <?= $count ?> people.
- If $count contains 3, result is I see 3 people.

Context Switching

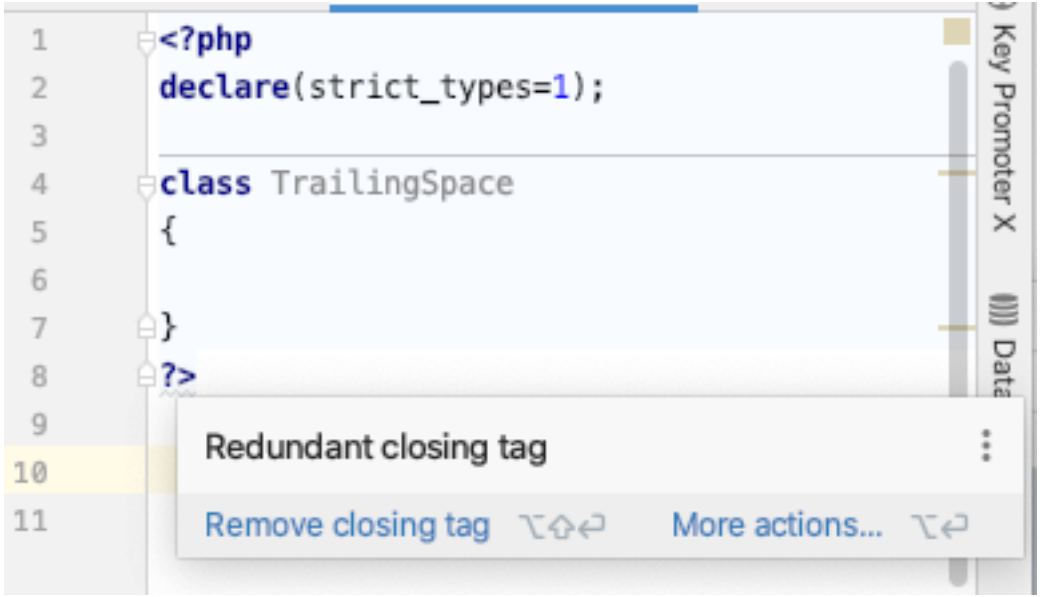
- PHP operates in two contexts:
  - Passthrough
  - PHP mode
- Passthrough mode is the default
- Text, usually HTML, is passed directly to the web server (or console output) from the PHP file

Context Switching

- <?php “open tag” starts it processing as PHP
- ?> means stop parsing PHP; go back to passthrough
- Important: Only use ?> closing tag if you have more text/html on the page!
- Trailing whitespace after ?> gets sent like any other text - if unintentional, produces “header already sent” errors
- Context switching has extremely low overhead

Example

It’s such a common problem that PhpStorm flags the error



Super-Basics

- PHP has a C-like syntax
- Statements are separated by semicolons (;)
- Blocks are enclosed by curly braces {}
- Three comment styles

```php
<?php
declare(strict_types=1);

/**
 * Class TrailingSpace
 * "Docblock" comment
 */
class TrailingSpace
{
    /* one-line comment */
    private $count = 0; // Inline comment
}
# Rarely-used "shell" style comment
```

Variables

- Begin with $
- Must start with letter (either case) or underscore
- Can contain letters, numbers, underscores
- Are assigned with the equals sign, called the assignment operator

```php
<?php
declare(strict_types=1);
$valid = 0;
$_valid = 0;
$Valid = 0;
$v4L1d_ = 0;

// Causes an error
$1nvalid = 0;

```

Naming Conventions

- Variables local to a given scope tend to be lower case and separated by underscores (“snake case”). $i, $j, etc., are common for counters.
- With Object Oriented Programming (OOP), CamelCase is more common:
  - Class names use initial cap, e.g., ThisClass
  - Variable and function names (called properties and methods in OOP) use initial lower case, e.g., thisValue or thisMethod().

Constants

- Constants by convention are upper-case words separated by underscores
- Often called “defines” because they are declared with the built-in define() function
- Value can be string or array

```
1  <?php
2  declare(strict_types=1);
3
4  define('MY_CONSTANT', 'some value');
5
```

Types and PHP

Types in PHP

- Problem:
  - HTTP is a text-based protocol
  - Data received from client is all text
- Solution:
  - PHP variables have no set type
  - Interpreter sets a type based on content and context (duck typing)
  - The rules WILL trip you up if you're not careful

Primitive Types in PHP

- PHP, like most scripting languages, uses inferred types for scalar variables
- This saves boilerplate (constantly converting from string to numeric, for example) but requires care
- Don’t do it! Use strict typing!

Primitive Types in PHP

- String - “this value”, ‘that value’ but use straight quotes (be careful about copy/paste from Word docs)
- Integer - 11235
- Float - 3.14159265358979
- Boolean - false
- Null - null

Type Juggling

- If you need to treat a scalar in a certain way, you can use typecasting by putting the type in parentheses(()).
- You can use var_dump() to get the inferred type

```php
<?php
declare(strict_types=1);

$string = '1';
var_dump($string); // string(1) "1"
$integer = (int) $string;
var_dump($integer); // int(1)
```

Integers and Type-Juggling

- Integers have a few important but tricky rules to remember when being cast from other inferred types
- Especially since some casting happens automatically when used in some contexts

Take the following values and assign them to a variable *while casting them to an integer.* Put a comment after each one predicting what the interpreter will store. Use `var_dump()` to reveal the answer after each one. Can you guess the rules the interpreter uses?

```
"My favorite number is 0."
"My favorite number is 1."
"2 of my favorite numbers are 0 and 1."
"2.1 of my favorite numbers are 0 and 1.1"
2.1
false
true
"true"
```

```php
<?php
$integer = (int) "My favorite number is 0.";
var_dump($integer); // int(0)
$integer = (int) "My favorite number is 1.";
var_dump($integer); // int(0)
$integer = (int) "2 of my favorite numbers are 0 and 1.";
var_dump($integer); // int(2)
$integer = (int) "2.1 of my favorite numbers are 0 and 1.1";
var_dump($integer); // int(2)
$integer = (int) 2.1;
var_dump($integer); // int(2)
$integer = (int) false;
var_dump($integer); // int(0)
$integer = (int) true;
var_dump($integer); // int(1)
```

Integer Casting Rules

- If a string starts with integer characters, anything after the first non-integer character will be discarded and the remaining (leftmost) characters will be converted to an integer
- Floats will drop the decimal and anything afterward
- True and false will be converted to 1 and 0, respectively

Variable Output

- In addition to the short variable tag `<?= $count ?>`, you can use print or echo to output scalar variables
- They are language constructs, not functions, and don't require parentheses
- Echo can take multiple variables separated by commas (,); therefore echo is used more often
- Both output strings no matter the inferred type

```php
<?php
$string = "1";
$integer = (int) $string;
echo $string, $integer; // 11
```

Built-in Functions, Strings, and Math

Oh my!

Function Basics

- PHP has a lot of built-in functions
- Always look for one first before writing your own
- Documentation at php.net/function_name
- Call function by name with parentheses, whether the function takes arguments or not

```
<?php
phpinfo(); // outputs info on PHP's config and environment
```

Function Basics

- Arguments are passed inside parentheses
- Separate multiple arguments with commas (,)
- The return value can be assigned to a variable or used with echo
- Functions can both produce output and return a separate value

```php
<?php
$string = "I love Ruby!";
$new_string = str_replace("Ruby", "PHP", $string);
echo $new_string; // I love PHP!
echo str_replace("Ruby", "PHP", $string); // I love PHP!
```

Quoting Strings

- Strings can be single-quoted or double-quoted (use straight quotes!)
- Double-quoted strings are interpolated

```php
<?php
$color = '$phrase = Green';
$phrase = "$color Eggs and Ham";
echo $phrase; // $phrase = Green Eggs and Ham
```

- Each quote type can have its own quote escaped for output

```php
<?php
echo 'It\'s a dog!'; // It's a dog!
echo "It says \"woof!\""; // It says "woof!"
```

Escape Codes in Strings

- Double-quoted strings can include line endings, tabs, or other characters by escape codes as well as literal $. These help format command line or <pre> output

```
<?php
echo "Home is where \$HOME is.\n";
echo "\t/my/home\n";
/*
%>php test.php
Home is where $HOME is.
   /my/home
%>
*/
```

Concatenation

- String concatenation is done with a period / full-stop (.)

```php
<?php
$string = 'Yellow' . ' +'; // Yellow +
$string = $string . ' Blue'; // Yellow + Blue
$string .= ' make Green';
echo $string;
// Yellow + Blue make Green
```

String Functions

- See php.net/manual/en/ref.strings.php for the full list

```
echo htmlspecialchars("Bacon & Eggs > Eggs & Cheese"), "\n";
// Bacon &amp;amp; Eggs &amp;gt; Eggs &amp;amp; Cheese
echo strtolower("MyEmail@example.com"), "\n";
// myemail@example.com
echo trim(" MyPassword "), "!\n";
// MyPassword!
echo nl2br("Break\nthe\nline."), "\n";
/*
Break<br />
the<br />
line.
*/
```

Basic Arithmetic

- Arithmetic operations work on integers and floats. PHP accounts for the differences automatically via duck typing rules. Order of operations and parentheses work as you'd expect
- Addition: echo 1 + 1.0; // 2
- Subtraction: echo 2.1 - 1; // 1.1
- Multiplication: echo 1 * 1; // 1
- Division: echo 1 / 0; // Divide by zero error
- Modulus: echo 5 % 2; // 1

Arithmetic Assignment Operators

- There are also operators to update the value on the left with an arithmetic operation between that value and the value on the right

```php
<?php
$apples = 5;

$apples += 2;               //  7 apples

$apples -= 5;               //  2 apples

$apples *= 4;               //  8 apples

$apples /= 2;               //  4 apples

$apples %= 3;               //  1 apple
```

Integer Increment and Decrement Operators

- $count++ means use the value before increment
- ++$count means increment before using the value

```php
<?php
$count = 1;
echo $count++; // 1
echo $count; // 2
echo ++$count; // 3
echo $count--; // 3
echo $count; // 2
echo --$count; // 1
```

Arrays

- Here's where PHP gets… different
- PHP arrays are technically ordered key-value maps
- Where other languages have stacks, queues, vectors, lists, arrays, hash maps, collections, etc., PHP puts them all into a single type - the PHP array
- Quite simple and efficient for most cases, tricky for edge cases
- Arrays can be assigned to variables, used as function arguments
- Naming rules same as for scalar variables, e.g., $myStuff

Array Structure

- Whether specified or not, PHP arrays are made up of zero or more key-value pairs that preserve the order of the pairs - though not all array operations will preserve that order!
- Keys can be integers or strings - booleans and floats are cast to integer
- Values can be any PHP value or data structure - scalars, arrays, objects
- Arrays can be multi-dimensional

Creating Arrays (old syntax)

```php
$numerically_indexed_array = array('Alice', 'Bob');
var_dump($numerically_indexed_array);
/* array(2) {
  [0]=>
  string(5) "Alice"
  [1]=>
  string(3) "Bob"
} */
$associative_array = array('teacher' => 'Alice', 'student' => 'Bob');
var_dump($associative_array);
/* array(2) {
  ["teacher"]=>
  string(5) "Alice"
  ["student"]=>
  string(3) "Bob"
} */
```

Creating Arrays (newer syntax, PHP 5.4+)

```php
<?php
$numerically_indexed_array = ['Alice', 'Bob'];
$associative_array = ['teacher' => 'Alice', 'student' => 'Bob'];
```

Accessing Array Elements

- Numerically-indexed arrays start counting from zero. Just put the index integer in square brackets.

```php
$array = ['Alice', 'Bob', 'Charley'];
echo $array[1]; // Bob
```

- Associative arrays use the key in square brackets. Multi-dimensional arrays simply keep adding brackets for each level

```php
$array = ['teacher' => 'Alice', 'students' => ['Bob', 'Charley']];
echo $array['students'][0]; // Bob
```

Adding Elements to an Array

```php
// Numerically-indexed:
$array = ['Alice', 'Bob'];
$array[] = 'Charlie'; // ['Alice', 'Bob', 'Charlie']

// Associative
$array = ['teacher' => 'Alice', 'students' => ['Bob', 'Charley']];
$array['aide'] = 'Daniel'; // adds a new element
$array['teacher'] = 'Ethel'; // replaces value for 'teacher' key
```

Array Functions

- About 80 array functions - use the online manual

```php
<?php
$ip_address = "127.0.0.1";
$ip_components = explode('.', $ip_address);
var_dump($ip_components); /*
array(4) {
  [0]=>
  string(3) "127"
  [1]=>
  string(1) "0"
  [2]=>
  string(1) "0"
  [3]=>
  string(1) "1"
} */
```

```php
<?php
$students = ['Alice', 'Bob', 'Charley'];
$class_list = implode(', ', $students);
echo $class_list; // Alice, Bob, Charley
```

Control Structures

And conditional logic

If - else if - else

```php
<?php
$condition = true;
$other_condition = false;

if ($condition) {
    echo '$condition is true';
} else if ($other_condition) {
    echo '$other_condition is true';
} else {
    echo 'Neither $condition nor $other_condition is true';
}
```

Ternary Conditional Operator

For very short if statements, PHP supports a ternary operator:

```
// do the opposite
($votes > $previous_votes) ? $votes-- : $votes++;
```

You can shorten it further if you only need to take action in the negative case:

```
// avoid uninitialized variable notice
isset($votes) ?: $votes = 0;
```

For readability reasons, use for the simplest expressions only. It's always OK to use the long form.

Switch

You may be less familiar with the `switch` statement, which allows branching on different values of a variable.

```
switch ($action) {
    case ‘upvote':
        $votes++;
    break; // important

    case 'downvote':
        $votes--;
    break;

    case 'clear_rating';
    default:
        $votes = $previous_votes;
}
```

Loops

Loops in PHP are fairly standard:

```php
<?php
$animals = ['lion', 'tiger', 'bear'];
for ($i = 0; $i < count($animals); $i++) {
    echo $animals[$i] . "\n";
}

// older PHP
while (list(, $animal) = each($animals)) {
    echo $animal . "\n";
}
reset($animals); // pointer is moved by while()

// newer PHP
foreach ($animals as $animal) {
    echo $animal . "\n";
} // no reset necessary; works on a copy
```

“Truthy” in PHP

If you come from a strictly-typed language, what PHP evaluates as `true` and `false` may surprise you.

`0`, `0.0`, `false`, `array()`, `""`, `"0"` and `null` ***all*** evaluate to `false`.

`true`, `1`, `array("")`, `-1`, `"-1"`, and `"false"` all evaluate to `true`.

This is a result of PHP’s type juggling as it attempts to compare values.

Comparison

The default comparison operators are very familiar for anyone used to C-like languages.

- Loose equivalence: ==, !=
- Mathematic: >, >=, <, <=, <>

PHP adds strict comparison operators that test type as well as content (no type juggling):

- Strict equivalence: ===, !==

User Functions in PHP

Function Creation

- User-defined functions are created with the function keyword, the name of the function, and parentheses, optionally including variables to hold arguments passed to the function
- Function code is enclosed in curly braces
- Functions can be defined anywhere in the code

```php
<?php
echo_string("Hello world!"); // Hello world!

function echo_string($string) {
    echo $string;
}
```

Function Naming

- Function name rules are the same as variables - begin with letter or underscore, followed by zero or more letters, underscores, digits
- Function declarations must include parentheses () at the end whether or not arguments are used

Default Values and Optional Arguments

You can provide a default value to function arguments. Doing so makes the function argument optional. Arguments with default values must always come **last** in the function signature.

```php
<?php
echo_string("Hello world"); // Hello world!
echo_string("Hello world", "."); // Hello world.

function echo_string($string, $second_string = "!") {
    echo $string, $second_string;
}
```

Recognizing a Superglobal

Almost done!

Superglobal Recognition

- Superglobals are always in all-caps $_COOKIE
- Superglobals usually begin with an underscore $_ENV
- You'll probably see and use $_SERVER
- List at https://www.php.net/manual/en/language.variables.superglobals.php
- They are dangerous!

Superglobals Are Dangerous

- The content in superglobals comes from outside your script/ code and should be considered suspect
- IP addresses, user-agent identifiers, accepts headers, are all user-supplied and can be spoofed or contain dangerous payloads
- Server environments can be compromised
- Practice defense-in-depth

Summary

- PHP syntax is similar to other C-like languages
- PHP usually receives a request from the web server, does its thing, and sends a response back to the web server
- Modern PHP software usually takes an Object Oriented design approach - that’s up next