Evozon PHP Internship

Prep course

PHP Syntax

A PHP script starts with **<?php** and ends with **?>**, unless it only contains PHP, in which case the closing tag should be omitted.

Variables [link]

Variables are named memory locations used for storing data.

```
$name = 'Eva';
```

Variable names in PHP start with **$** and are case-sensitive.

A valid variable name starts with a letter or underscore, followed by any number of letters, numbers, or underscores.

You should be aware that, while it is perfectly legal to name your variables $a or $_b, it is advisable that you take time to come up with accurate and descriptive names for them. For example, if you are measuring something, it could be helpful to specify the units ($temperatureCelsius, $intervalSeconds). You might also want to make it clear what state the data is in at a certain point in your script ($unsanitizedContent). The greater the scope of the variable, the more important it is that its name explicitly identifies its content.

Additionally, PHP provides a number of predefined variables.

Variable scope [link]

Where in your script you declare your variable will determine its accessibility. The domain of a variable's accessibility is called its **scope**. From a scope perspective, in PHP, variables can be of the following types:

- local variables
- function parameters
- global variables
- static variables

As a rule, you cannot access a variable declared in a scope more local than the current one.

If you attempt to access a variable outside its scope, you will get a notice: "Undefined variable $x on line $y".

A variable defined within a function is a **local variable**, meaning it can only be referenced inside that function. Upon exit from the function, the variable is destroyed.

Function parameters are similar to local variables, in that their scope is also restricted to that of the function (unless they are passed by reference). They are also destroyed upon exit from the function.

Global variables can be accessed anywhere in the program. To change a global variable from inside a function, you must prepend the word **global** to the variable you are accessing. For example:

```
$counter = 1;

function incrementCounter()
{
    global $counter;
    $counter++;
}

incrementCounter(); // $counter will be equal to 2
```

An alternative way to declare a global variable is to add it to the $GLOBALS array.

Static variables do not lose their value when the function exits and still hold that value if the function is called again.

```
function incrementCounter()
{
    static $counter = 0;
    $counter++;
}

incrementCounter(); // $counter is equal to 1
incrementCounter(); // $counter is equal to 2
```

Constants [link]

Constants hold values that do not change during the execution of a program. They may start with a letter or underscore, are case sensitive, contain only alphanumeric characters and underscores. Constants have global scope.

```
define('SPEED_OF_LIGHT_MS', 299792458);

const SPEED_OF_LIGHT_MS = 299792458; // preferred
```

PHP also provides predefined constants and a set of magic constants as part of its core.

Data types and operators

Data types [link]

The data type of a variable is automatically set by the PHP parser and it depends on the value assigned to the variable.

```
$a = 1; // int type
$b = 1.5; // float type
```

The value of a variable can be evaluated differently depending on the context:

```
// float type evaluated as string type
echo $b; // "1.5"
```

Although PHP is a loosely typed language, you should be aware of the available data types. These are:

Type	Classification	Description
int	scalar	integer (whole number - base 10, base 8, base 16)
float	scalar	floating-point number (with fractional parts)
string	scalar	series of characters
bool	scalar	boolean value (true / false)
array	composite	ordered map
object	composite	user-defined type
resource	special	external resource
null	special	no value
callable	special	function/method

Operators [link]

A frequently used operator is the **assignment operator** (**=**). Typically, if you are assigning the value of a variable to another variable, the assignment will be by value, not by reference. If you wish to assign a value by reference, you must use the following syntax:

```php
$a = 3;
$b = &$a;

$b = 2;

echo $a; // 2
```

The most common **arithmetic operators** are:

- + The addition operator
- - The subtraction and negation operator
- * The multiplication operator
- / The division operator
- %, the modulus operator, returns the remainder

Typical shorthand for addition and subtraction:

- + = adds a value to the variable: $a + = 1 adds one to the variable $a.
- -= subtracts a value from the variable: $a -= 1 subtracts one from the variable $a.

Type juggling [link]

Casting to type [link]

To cast a variable to a type, you write the type name in parentheses before the variable:

```php
$price = (int) 15.8; // 15

$price = "This is a price";
$price = (int) $price; // 0

$price = 15.8;
$price = (array) $price; // [0 => 15.8]
```

As we were saying previously, in PHP, variables are sometimes automatically cast to best fit their context. For example:

```php
// adding an integer and a string
$a = 1;
$b = "5";
$c = $a + $b; // 6

// conversion of an integer to boolean
$a = 1;
```

```php
if ($a) {
    // conversion of an integer to string through interpolation
    echo "It is entirely the case that {$a}";
   // this will execute and will output "It is entirely the case that 1"
}
```

Checking type [link]

To check a variable's type, you use the **gettype()** function.

To check whether a variable has a specific type, you use the **is_{{type}}()** functions (e.g. **is_null()**, **is_bool()**, etc).

Comments: syntax and practice

Comments are used to insert notes into the code. They have no effect on the parsing of the script.

PHP uses the two standard C++ notations for single-line (//) and multiline (/* */) comments. The Perl comment notation (#) may also be used to make single-line comments.

```php
<?php
// single-line comment

# single-line comment

/* multi-line
comment */
?>
```

How to use comments: 5 practical tips

1. Don't comment on obvious facts:

```php
// Increase counter
$i++;
```

2. Use explicit names for variables and functions instead of clarifying with comments.

```php
/**
* Explaining what a counter is counting is a good idea :)
*/

// line count
```

```
$i = 0;

/**
* ... but giving the counter an explicit name is better.
*/

$lineCount = 0;
```

3. Explain why you used a particular approach in your function.

```
// this algorithm was chosen because it is X times faster than algorithm Y.
```

4. Explain why your constants have the values they do.

```
// reasonable pagination limit - humans will usually not visit more than 5
pages anyway
const MAX_PAGE = 1000;
```

5. Add // TODOs to mark places in your code that you need to go back to.

```
// TODO: only allow alphanumeric IDs.
// TODO: sort query results.
```

Control structures [link]

An **if** statement is executed when the condition inside the parentheses is evaluated to true. Some non-boolean values will be converted to booleans in this context.

```
if ($a == 1) {
    return "{$a} is equal to 1";
}
```

When using **if** statements, avoid writing superfluous code like the following:

```
if ($a == $b) {
    return true;
}
```

As the result of the comparison is already a boolean, it can be returned directly:

```
return $a == $b;
```

If statements can be extended with any number of **elseif** clauses and/or a final **else** clause.

```
if ($grade == 10) {
    echo "I win at life, the Universe, and everything!";
} elseif ($grade > 8) {
    echo "I'm an ok student.";
} elseif ($grade > 4) {
    echo "At least I'm passing?";
} elseif ($grade <= 4) {
    echo "Not doing great ...";
} else {
    echo "We neither cover nor care about your weird grading system.";
}
```

Avoid using **elseif** / **else** clauses unnecessarily, like in the following example:

```
if ($a == $b) {
    return 0;
} else {
    // do stuff
}
```

The above **else** clause only adds needless nesting and should be omitted:

```
if ($a == $b) {
    return 0;
}

// do stuff
```

The **switch** statement matches an integer, float, or string value with a sequence of **cases,** passing execution to the corresponding case. A **switch** can contain any number of case clauses, which may or may not end in a **break** instruction, and may also end with a **default** label for handling unspecified cases.

```
$month = date("M");

switch ($month) {
    case "Oct":
        echo "It's October. I hope you're ready for Halloween!";
        break;
```

```
    case "Dec":
        echo "It's December. Have you been good?";
        break;
    default:
        echo "This must be some other month. What a sad world it is.";
        break;
}
```

It is also possible to use an alternative syntax for control structures. This is typically used when mixing PHP with HTML, as it improves readability.

In PHP we also have the **ternary operator** (?:), which can replace a single if/else structure. This operator takes three expressions. If the first one is true, the second expression is returned, otherwise the third one is returned.

```
return ($a == $b) ? "{$a} is equal to {$b}" : "{$a} is not equal to {$b}";
```

Including other source files

Code that must be called in more than one place can be isolated in a separate file and subsequently included in all relevant places using the **include** statement. As this statement is a special language construct, not a function, parentheses should not be used.

```
include "file.php";
```

The given file path can be relative, absolute, or omitted entirely. A relative path will be relative to the location of the file containing the **include** statement. An omitted path means that the directory of the file containing the **include** statement will be used. If the path is relative or omitted and the file cannot be found in the aforementioned locations, **include** will attempt to locate the file in the directories specified in the php.ini **include_path** directive.

Other language constructs for importing files:

- **require**: when a file import fails, **require** will throw an error and stop script execution, while **include** will only throw a warning and allow execution to proceed. The import can fail either because the file is missing or the user running the web server does not have read access to it.
- **include_once**: if the file has already been included, it will not be included a second time.
- **require_once**: identical to **include_once**, but for the **require** construct.

Use cases: the header, navigation, or footer of your web application.

Language constructs: echo, print

echo and **print** are both language constructs used to output scalar values to the screen.

echo does not have a return value, while **print** always returns 1. Additionally, **echo** is capable of outputting multiple strings.

```
echo "don't", "worry", "be", "happy";

print "That is not dead which can eternal lie, and with strange aeons even death may die."
```

Other options for outputting data: printf, sprintf

You use the **printf()** statement when you want to output a mix of static and dynamic information. An example usage would be:

```
printf("This year we have %d interns", 6);
```

Here, **%d** is a type specifier, and represents a placeholder. The **d** means that an integer will replace it. When the statement executes, the argument will be inserted into the string. To ensure correct behaviour, remember to comply with the specification and pass it an integer (if a float is passed, it will be rounded down to the closest integer, and 0 will be output instead of strings).

sprintf() is nearly identical to **printf()**, but it assigns the data to a string instead of outputting it directly.

```
$a = sprintf("There is a %s on the doorstep", "thing");
```

For a discussion of the available options for the format string, please consult the PHP manual.

Other useful functions

isset, empty, unset

isset() is used to check whether a variable is set and is not null.

```
isset($a); // false

$a = null;
isset($a); // false
```

empty() checks whether a variable is empty, i.e. if it does not exist or its value evaluates to false.

unset() deletes a variable from the current scope. Unsetting a variable differs from setting a variable to null.

```
$a = null;
echo "{$a}"; // will execute (and output nothing :) )

unset($a);
echo $a; // will throw a notice that $a is undefined
```

die, exit

die() and **exit()** are equivalent constructs, used to output a message and terminate the running script.

Functions [link]

Functions are named pieces of code that can be invoked repeatedly in your script. In PHP, a function is defined as follows:

```
function greet()
{
    return "Hello";
}

$a = greet(); // $a is "Hello";
```

Function names are case-insensitive, but it is good practice to call the function in the same casing you used for its definition.

Functions parameters [link]

Within the parentheses that follow the function name, zero or more parameters can be specified, allowing a corresponding number of arguments to be passed to the function.

Default parameters

You can assign default values to parameters by specifying these values in the function declaration. If you want some parameters to have default values, they must be declared to the right of those that do not have default values.

```
function greet(string $firstname, $lastname = '')
{
    return trim("Hello, {$firstname} {$lastname}");
}

echo greet("Randolph"); // Hello, Randolph
echo greet("Randolph", "Carter"); // Hello, Randolph Carter
```

Variable number of parameters [link]

You cannot call a function with fewer arguments than its parameters list (except for parameters that have a default value), but you can call it with more. To get the arguments one at a time, you use the **func_get_arg()** function.

```
function greet()
{
    $a = func_get_arg(0);
    $b = func_get_arg(1);
    $c = func_get_arg(2);
```

```
    return "Hello, {$a}, {$b}, and {$c}";
}

echo greet("Cthulhu", "Yog-Sothoth", "Azathoth"); // Hello, Cthulhu,
Yog-Sothoth, and Azathoth
```

You can also retrieve all the arguments with the **func_get_args()** function and get the number of passed arguments with **func_num_args()**.

As of PHP 5.6, you can use the **...** token to indicate that a function accepts a variable number of arguments. This parameter behaves like an array and must be specified last in the list.

```
function greet(...$names)
{
    $greeting = "Hello";

    foreach ($names as $name) {
        $greeting .= ", {$name}";
    }

    return $greeting;
}

echo greet("Walter", "William", "Henry"); // Hello, Walter, William, Henry
```

Type hints/declarations

To force a parameter passed to a function to be of a certain type, you can add the type name before the parameter. Beware: PHP versions lower than 7.0 do not support type hinting with scalar data types.

```
function greet(string $name)
{
    // ...
}
```

It is advisable to use type declarations whenever possible.

You can also type hint all the members of a packed argument: function greet(string …$names). This is great for enforcing a type to all the arguments of a function.

Return statement [link]

The **return** statement stops the execution of the current function and returns control back to the point (and implicitly scope) from which the function was called.

If **return** is called from the global scope, the script execution is halted.

If a function returns a value, the function call will evaluate to that value.

A function without a return value will return null.

Starting from 7.0, PHP supports return type declarations.

```
function greet(string $name) : string
{
    return "Hello, {$name}";
}
```

It is advisable to use return types whenever possible.

Passing arguments by reference [link]

You might at some point want to make changes to function arguments that persist outside the function's scope. In this case, you will want to receive the arguments by reference. To achieve that, you should prepend an ampersand to the argument.

```
function incrementByOne(&$int)
{
    $int++;
}

$a = 0;
incrementByOne($a); // $a is 1
```

Strings [link]

When specifying a string value, you can enclose it in single or double quotes:

```
$a = 'string';
$b = "also a string";
```

Single-quoted strings are literal. However, if you use double quotes, you can include a variable name in your string and it will be substituted for its value (variable interpolation). For example:

```
$a = 7;
echo "My lucky number is {$a}"; // My lucky number is 7
```

Surrounding your variables with curly braces is optional but encouraged if the variables are scalars. However, if you want to use arrays or objects within double-quoted strings, the braces are mandatory:

```
$user = new User();
$array = ['key' => 'value'];

echo "User email: {$user->getEmail()}"; // ... hopefully you've implemented
this method and it returns the user's email :)

echo "The value for 'key' is {$array['key']}"; // The value for 'key' is
value
```

Additionally, you can use escaped characters like **\n** in a double-quoted string, and they will be interpreted as new lines.

As an alternative to the single and double quotes, you can use the nowdoc and heredoc syntax, respectively. These are used when you need to define strings that span multiple lines.

```
// nowdoc
echo <<<'IDENTIFIER' // notice the single quotes
Hello World.
IDENTIFIER;

// heredoc
echo <<<IDENTIFIER
Hello World.
IDENTIFIER;
```

The **nowdoc** opens with the <<< operator, followed by a single-quoted identifier and a new line. The enclosed string is followed by a new line containing the identifier, which marks the end of the string. Variables are not parsed inside a nowdoc string, as they are not parsed within single-quoted strings.

The **heredoc** syntax differs from the nowdoc by the fact that the identifier it opens with is not quoted. Variables are parsed within a heredoc string, as they are within double-quoted strings.

If you want to compare strings, you can use:

- the comparison operator (==);
- the strict comparison operator (===);
- the case-sensitive function **strcmp()**;
- the case-insensitive function **strcasecmp()**.

A character in a string can be retrieved by its index:

```
$str = 'string';
echo $str[0]; // s
```

Arrays [link]

Basics

Declaring an array:

```
$array      = array(); //obsolete
$alsoArray = [];
```

Casting to array:

```
$string           = 'string';
$arrayString      = (array) $string; // [0 => $string]

$int              = 3;
$arrayInt         = (array) $int; // [0 => 3]

$bool             = true;
$arrayBool        = (array) $bool; // [0 => true]

$object           = new stdClass();
$object->property = 'value';
$arrayObj         = (array) $object; // ['property' => 'value']
```

Be aware that you can use an object as an array if your class implements the ArrayAccess interface.

An array is like a box of chocolates:

```
$a = [7, 1 => 3.14, 'two' => 'we need strings too', new stdClass(), ['PHP',
'is', 'great']];

/**
[
    [0] => 7
    [1] => 3.14
    [two] => we need strings too
    [2] => stdClass Object ()

    [3] => Array
        (
            [0] => PHP
            [1] => is
            [2] => great
```

```
        )
]
*/
```

As you can see, it is possible to have values of different types and keys that are both integers and strings.

If you use a float as a key, it will be cast to an integer.

```
$a[7.5] = 'value'; // [7 => 'value']
```

If you use a boolean as key, it will also be cast to an integer.

```
$a[false] = false; // [0 => false]
$b[true]  = true; // [1 => true]
```

If you use null as key, you will be judged harshly.

```
$a[null] = 'value'; // [[] => 'value']
```

Arrays and objects can't be used as keys.

Accessing array elements

Array elements can be accessed with the square brackets syntax.

```
$a = ['element 0', 'element 1', 'element 2'];
$b = $a[0];  // 'element 0'
```

Changing array elements

Array elements can be changed by using the square brackets syntax.

```
$a = ['key' => 'value'];
$a['key'] = 'new value'; // ['key' => 'new value']
```

Elements can be removed from an array with **unset()**.

```
$a = ['key' => 'value'];
```

```
unset($a['key']); // []
```

Calling unset() on an array will delete the entire array.

Array operations

Adding an element at the end of an array

A lot of times, you will use an array as a collector variable, where you will store some elements processed during an iteration. In order to easily add an element at the end of an array, PHP provides a syntactic sugar construct:

```
$powers = [];

for ($i = 1; $i <= 10; $i++) {
    $powers[] = $i * $i;
}
```

The code above creates the following array in $powers:

```
[
    [0] => 1
    [1] => 4
    [2] => 9
    [3] => 16
    [4] => 25
    [5] => 36
    [6] => 49
    [7] => 64
    [8] => 81
    [9] => 100
]
```

Summing up arrays

The plus operator (+) is allowed for array variables and the array value it outputs follows the next rules:

1. takes the keys and elements from the first array;
2. takes the keys not found in the first array from the second array, together with their values;
3. returns a result composed of the key => value pairs found in step 1 and step 2.

To put this differently, the operation basically creates the reunion of the keys and then populates it with values from the two array, favoring the first one.

Example:

```
$a = [
    0   => 1,
    'a' => 2,
    1   => 3
];

$b = [
    0   => 3,
    'a' => 4,
    'b' => 2,
    1   => 1
];


$c = $a + $b;
```

After running the above lines of code the value of $c will be:

```
[
    [0] => 1
    [a] => 2
    [1] => 3
    [b] => 2
]
```

Merging arrays

In PHP there is also the notion of merging two arrays into one. This operation is different to addition and is governed by the following rules:

- the first array is used as a base for the result;
- the second array is iterated element by element and:
    - if the element key is an integer, then the value is appended to the result array, using the next integer key available.
    - if the element key is a string, then
        - if the string key is already found in the first array, the initial value is replaced with the currently iterated value;
        - if the string key is unique so far, the element is added associated to the key to the result array.

Example:

```
$a = [
    0   => 1,
    'a' => 2,
    1   => 3
];

$b = [
    0   => 3,
    'a' => 4,
    'b' => 2,
    1   => 1
];


$c = array_merge($a, $b);
```

The value of $c after the merge will be:

```
[
    [0] => 1
    [a] => 4
    [1] => 3
    [2] => 3
    [b] => 2
    [3] => 1
]
```

Using arrays as stacks

Given the fact that the array type is the de facto structured data type in PHP, it is versatile enough to allow usages that in other programming languages would need dedicated data types.

Of course, there are OOP implementations for stacks in PHP, in the form of later added extensions (see SPL Stack or the newer DS Stack), but the first thing at hand when you are in need of a stack is the plain old array.

You can make an array act like a stack in two ways:

- by pushing and popping values at the end of an array
- by unshifting and shifting values at the beginning of an array

Stack with array_push() and array_pop() (FIFO)

```
$stack = [];
```

```php
// pushing three values on the stack
array_push($stack, 1);
print_r($stack);

array_push($stack, 2);
print_r($stack);

array_push($stack, 3);
print_r($stack);

// popping three values from the stack
print array_pop($stack);
print array_pop($stack);
print array_pop($stack);

print_r($stack);
```

Given the above snippet of code the output of it will be:

```
Array ( [0] => 1 )
Array ( [0] => 1 [1] => 2 )
Array ( [0] => 1 [1] => 2 [2] => 3 )
321
Array ( )
```

Stack with array_unshift() and array_shift() (LIFO)

A similar approach is also possible using the array_unshift/array_shift pair of functions:

```php
$stack = [];

// pushing three values on the stack
array_unshift($stack, 1);
print_r($stack);

array_unshift($stack, 2);
print_r($stack);

array_unshift($stack, 3);
print_r($stack);

// popping three values from the stack
print array_shift($stack);
print array_shift($stack);
print array_shift($stack);

print_r($stack);
```

The output for the above code is:

```
Array ( [0] => 1 )
Array ( [0] => 2 [1] => 1 )
Array ( [0] => 3 [1] => 2 [2] => 1 )
321
Array ( )
```

Using arrays as queues

The same as with stacks, if you need a quick queue behaviour in your algorithms, then your best choice is an array.

There are OOP implementations for queues that should be considered for production code such as DS Queue or SPL Queue.

Example queue behavior with array:

```
$queue = [];

// enqueuing three values
array_push($queue, 1);
print_r($queue);

array_push($queue, 2);
print_r($queue);

array_push($queue, 3);
print_r($queue);

// dequeuing three values
print array_shift($queue);
print_r($queue);
print array_shift($queue);
print_r($queue);
print array_shift($queue);
print_r($queue);
```

The output of the above snippet is:

```
Array ( [0] => 1 )
Array ( [0] => 1 [1] => 2 )
Array ( [0] => 1 [1] => 2 [2] => 3 )
1
Array ( [0] => 2 [1] => 3 )
```

```
2
Array ( [0] => 3 )
3
Array ( )
```

Conversions to other types

To convert arrays to other data types, you can do the following:

```php
$a = [
    0   => 1,
    'a' => 2,
    1   => 3
];

// you cannot cast directly to string, but you can use the implode()
function
implode(', ', $a);
// $a is a string: "1, 2, 3"

(int) $a;
// $a is an int: 1

(object) $a;
/**
$a is an object:
object(stdClass)#1 (3) {
  ["0"]=>
  int(1)
  ["a"]=>
  int(2)
  ["1"]=>
  int(3)
}
*/
```

Files

There are situations during the development of web applications when you need to fallback to basic file reading or writing, even though you are using a framework that abstracts most of the low level code needed to access and change files data. Even if the framework of choice offers good wrappers around the basic file manipulation functions, as a programmer, you should be aware of the low level underpinnings of those wrappers.

Opening a file

Before you can do anything with a file's contents, you need to open it. Opening a file returns a **file handler** you'll have to later use in every file related function.

Similarly to C, you have to provide a **mode** when you open a file. The mode determines what kind of operations you are allowed to perform on the file, the position of the read/write pointer inside the file at open time, and the way opening an nonexisting file will behave.

For example, the "r" mode opens a file for read only and places the pointer for reading at the beginning of the file. You would use this mode when all you need to do with the file is read its contents end-to-end.

If you need to rewrite the contents of a file or create a new file with the given name, then you will use the "w" mode. This erases any content in an existing file or creates a new one if no file is found with that name.

In case you only want to append to a file that contains valuable content so far, you will open the file using the "a" (append) mode. This mode places the pointer for writing at the end of the file, so first time you will write something it will be appended.

There are also other modes that you are encouraged to explore here.

Example:

```
$filename = "plain.txt";
$handler = fopen($filename, "r");

if ($handler === false) {
     echo "Error in opening file";
     exit;
}

// do work with the opened file

fclose($handler);
```

Please keep in mind an opened file needs to be closed after you finished working with it. The function to close a file handler is **fclose($handler)**.

Reading from a file line by line

Example:

```
$filename = "plain.txt";
$handler = fopen( $filename, "r" );

if ($handler === false) {
     echo "Error in opening file";
     exit;
}

while(!feof($handler)){
    $line = fgets($handler);
    echo $line;
}

fclose($handler);
```

Please note that the **fgets()** function reads line by line based on the line termination characters (new line characters) and the returned string contains also the line termination character.

Reading from a file using a buffer - binary safe read

In order to exemplify the binary safe reading, we will provide an example of a snippet of code that concatenates the contents of two files - it appends at the end of a file the contents of a second file, and it does that in 1024 bytes chunks.

```
$fileOne = "a";
$fileTwo = "b";
$one = fopen($fileOne, "a");
$two = fopen($fileTwo, "r");

while(!feof($two)){
    $buffer = fread($two, 1024);
    fwrite($one, $buffer);
}

fclose($one);
fclose($two);
```

The script expects two files named “a” and “b” in the current directory. It opens both and appends the contents of “b” to “a” 1024 bytes at a time. Please note that this concatenation is

binary safe, the code moves streams of bytes and makes no assumptions about the files as being plain text files or having line endings.

OOP syntax [link]

To declare a class, the **class** keyword is used, followed by the class name and a set of curly braces. The naming convention for classes is title case.

To generate an instance of a class, the **new** keyword is used.

```
class Product
{
    //
}

$product = new Product(); // object(Product)#1 (0) {}
```

The class body can contain:

- **constants**;
- **properties** (fields/attributes);
- **methods**.

Properties and methods should have an access level specified. The available access levels are the following;

- **public**, which allows unrestricted access to the property/method;
- **protected**, which allows subclasses access to the property/method;
- **private**, which does not allow access to the property/method from anywhere outside the class.

To access methods and properties from inside the class, the **$this** pseudo variable is used, along with the arrow operator (**->**). The $this variable is a reference to the current instance of the class and can only be used within an object context.

Classes also have **constructor** methods, automatically called when an object is created. The constructor method is called **__construct()**.

```
class Product
{
    protected $name;

    public function __construct(string $name)
    {
        $this->setName($name);
    }

    public function setName(string $name) : Product
    {
```

```
        $this->name = $name;
        return $this;
    }

    public function getName() : string
    {
        return $this->name;
    }
}

$product = new Product('Harry Potter and the World of PHP 7');

/**
* object(Product)#1 (1) {
*   ["name":protected]=>
*   string(34) "Harry Potter and the World of PHP 7"
* }
*/


$name = $product->getName(); // $name is 'Harry Potter and the World of PHP
7'

$name = $product->name; // Uncaught Error: Cannot access protected property
Product::$name
```

You can also access methods and properties in the context of a class, not only in that of an object. These methods and properties are static and are declared with the **static** keyword.

```
class Debug
{
    static function printMessage(string $message)
    {
        echo "{$message}";
    }
}

Debug::printMessage("You got debugged!");
```

A class may be subclassed by using the keyword **extends**.

```
class Book extends Product
{
   //
}
```

A child class can override the properties and methods of the parent class by redeclaring them.

A class may also implement zero or more interfaces. To declare an interface, you use the **interface** keyword. To implement an interface, you use the **implements** keyword.

```
interface DiscountableInterface
{
    public function getDiscountedPrice() : float;
}

class Product implements DiscountableInterface
{
    public function getDiscountedPrice() : float
    {
        // obtain / calculate the discounted price
        return (float) $discountedPrice;
    }
}
```

A class takes on the type of the interface it implements.

```
$product = new Product("Alice's Adventures with Regex");

is_a($product, "DiscountableInterface"); // true
```

PHP also provides predefined interfaces, such as JsonSerializable.

A class can also use one or more **traits**. A trait contains methods that will become available to all the classes that use it. To declare a trait, you use the **trait** keyword. To use a trait, you use the **use** keyword.

```
trait CalculatorTrait
{
    // utilities for performing complex calculations
}

class Product implements DiscountableInterface
{
    use CalculatorTrait; // all utilities are now available to Product

    public function getDiscountedPrice() : float
    {
        // use CalculatorTrait methods to calculate discount
        return (float) $discountedPrice;
```

```
    }
}
```

Additional resources for web-related topics

- What happens when you click a link in the browser?