Server-side
Web Development

Unit 02. PHP basics. Strings, dates and arrays.

Fidel Oltra, Ricardo Sánchez

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2024-25

Contents

1 PHP language

PHP is an recursive acronym for ***PHP Hypertext Prepocessor***. PHP is a widely used open source scripting language for web delevopment that works on the server-side (back-end). The best PHP feature for web delevoping is that PHP code can be embedded into HTML tags.

W3Techs reports that *"PHP is used by 75.8% of all the websites whose server-side programming language we know."* PHP version 7.4 is still the most used version.

W3Techs server-side languages statistics

W3Techs PHP statistics

1.1 PHP versions

PHP, of course, has been improving and adapting since it was conceived in 1994.

PHP Version	Release Date	Support EOL
5.6	28 August 2014	31 December 2018
7.3	28 November 2019	6 December 2021
7.4	28 August 2014	28 November 2022
8.0	26 November 2020	26 November 2023
8.1	25 November 2021	31 December 2025
8.2	8 December 2022	31 December 2026
8.3	23 November 2023	31 December 2027
8.4	21 November 2024	31 December 2028

1.1.1 PHP 7.x

PHP 7 was the next version after **PHP 5.6**.

PHP 7.x versions brought enormous improvements in PHP engine performance. It also introduced a variety of new features that made it quickly adopted.

On **November 28, 2022**, the last PHP 7 version, PHP 7.4, went end of life.

Wikipedia: PHP 7

1.1.2 PHP 8.x

PHP 8 was released on November 26th, 2020. This version is a **JIT (Just-in-time) compiler**, that can provide substantial performance improvements for some use cases. Another notable changes are the addition of the match expression, types and other new features.

PHP 8.1 was released on November 25, 2021. It included several improvements, such as enumerations (also called "enums"), readonly properties and array unpacking with string keys.

PHP 8.2 was released on December 8, 2022. It includes a number of new features and improvements, such as Improved Type Variance, readonly classes, a new random extension and various performance improvements.

PHP 8.3 has been released on August, 29 2024. According to the website "It contains many new features, such as explicit typing of class constants, deep-cloning of readonly properties and additions to the randomness functionality. As always it also includes performance improvements, bug fixes, and general cleanup". It also includes a very useful function to validate a Json string, **`json_validate()`**;

Wikipedia: PHP 8

What's new in PHP 8.2

Php 8.3 released

1.2 PHP first script

Let's see an example. Imagine that we have a folder **`curs2425`** in our Apache root folder (in Windows `c:\xampp\htdocs`, in Linux `/var/www` or `/var/www/html`). Now let's create a file named `ud02-01.php` with this content:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example</title>
    </head>
    <body>
        <h2>
        <?php
            echo "Hi, I'm a PHP script!";
        ?>
        </h2>
    </body>
</html>
```

The file will be saved in the `curs2425` subfolder. Now, if we type in our browser

`http://localhost/curs2425/ud02-01.php`

we will get this result:

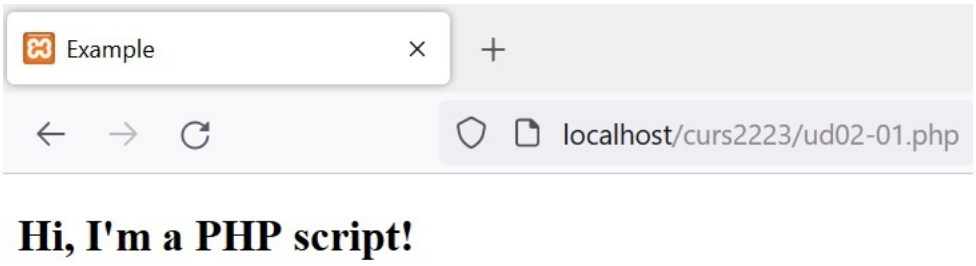

Figure 1: My first PHP script

> A file with PHP content always needs to have the **.PHP** extension at the end of the name.

As we can see in the image, the content generated by the PHP code has been merged with the HTML code to create a final content that our browser can understand and display on the screen. The PHP code embedded in the HTML code must be enclosed between the `<?php` and `?>` tags.

```php
<?php
echo "Hi, I'm a PHP script!";
?>
```

The PHP command `echo` outputs the given content to the screen. Actually, `echo` transforms the PHP content into HTML to be merged with the original HTML content.

2 Basic PHP syntax

Every PHP instruction ends with a semicolon (`;`).

```php
$variable="Hello, world";
```

2.1 Keywords

These words have special meaning in PHP. The following words cannot be used as variables, constants, class names, or function names:

Don't try to learn them all now! We'll learn the most important ones gradually.

PHP Keywords

__halt_compiler()	abstract	and	array()	as
break	callable	case	catch	class
clone	const	continue	declare	default
die()	do	echo	else	elseif
empty()	enddeclare	endfor	endforeach	endif
endswitch	endwhile	eval()	exit()	extends
final	finally	fn (as of PHP 7.4)	for	foreach
function	global	goto	if	implements
include	include_once	instanceof	insteadof	interface
isset()	list()	match (as of PHP 8.0)	namespace	new
or	print	private	protected	public
readonly (as of PHP 8.1.0) *	require	require_once	return	static
switch	throw	trait	try	unset()
use	var	while	xor	yield
yield from				

Figure 2: List of PHP keywords

2.2 Comments

Comments are used to insert notes into the code. They have no effect on the parsing of the script. PHP has two standard notations for comments:

- single-line comments: `//` or even the PERL notation `#`
- multiline comments: `/*` at the beggining of the coment, `*/` at the end.

```php
<?php
// this is a single-line comment
# this is another single-line comment
echo "Hi, I'm a PHP script!";
/* this is
   a multiline
   comment */
?>
```

Symbol	Name
/	Slash
//	Double slash
"	Quotation mark
#	Hash
-	Dash
.	Dot
?	Question mark
*	Asterisk

Figure 3: Some punctuation marks in English

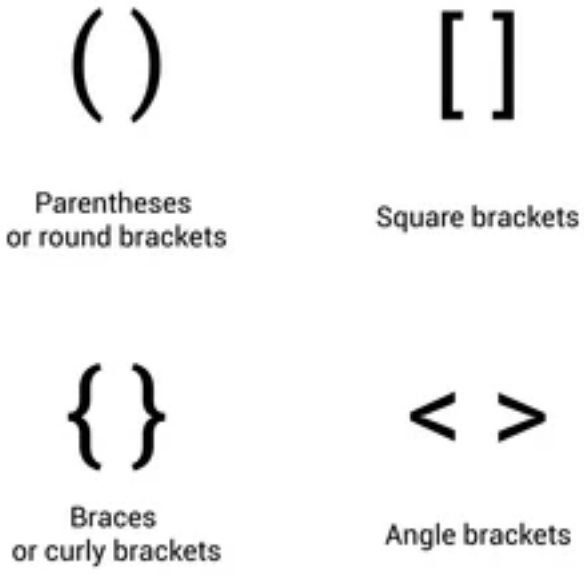

Figure 4: Types of brackets

2.3 PHP tags

When PHP parses a file, it looks for **opening** and **closing tags**, which are **<?php** and **?>** which tell PHP to start and stop interpreting the code between them. Parsing in this manner allows PHP to be embedded in HTML documents, as everything outside of a pair of opening and closing tags is ignored by the PHP parser.

PHP includes a **short echo tag <?=** which is a short-hand to the more verbose <?php echo.

Short tags, **<?** and **>** (example three), are available by default but can be disabled either via the ***short_open_tag*** **php.ini** configuration file directive, or are disabled by default if PHP is built with the ***–disable-short-tags*** configuration.

```
//Example 1
<?php echo 'if you want to serve PHP code in XHTML or XML documents, use
 ↪  these tags'; ?>

//Example 2
You can use the short echo tag to <?= 'print this string' ?>.
It's equivalent to <?php echo 'print this string' ?>.

//Example 3
<? echo 'this code is within short tags, but will only work '. 'if
 ↪  short_open_tag is enabled'; ?>
```

> As short tags can be disabled, it is recommended to use only the normal tags (and <?= ?>) to maximise compatibility. In this course we will use preferably the standard **<?php** and **?>** tags.

If a file contains only PHP code, it is preferable to omit the PHP closing tag at the end of the file. This prevents accidental whitespace or new lines being added after the PHP closing tag, which may cause unwanted effects.

```
<?php
echo "Hello world";

// ... more code

echo "Last statement";

// the script ends here with no PHP closing tag
```

A PHP file can have multiple PHP tags merged with HTML content. The variables defined in a PHP tag already closed are visibles in the following tags.

```
<?php $a = "world" ?>
<p> Hello
<?php echo $a ?>
</p>
```

3 Variables

In PHP language, a variable name starts with a dollar sign ($) followed by the identifier of the variable. PHP is a loosely typed language, so we don't need to specify the data type of the variable.

```
$variable = 1;
```

It is not necessary to initialize variables in PHP, however it is a very good practice.

PHP strings can be delimited in some different ways. There are two common notations: double quote (" ") and single quote (' '). Variables within double quotes are changed by their value. Variables within single quotes are not changed by their value and their name will be literally shown.

```
<?php
$name="John";
echo "Hello, $name <br/>"; // Will show Hello, John
echo 'Hello, $name <br/>'; // Will show Hello, $name
?>
```

Keep in mind that variable names are **case sensitive**. They can include underscores, characters and numbers, but they cannot start with a number. They also cannot contain spaces or special characters, and the identifier must not be a reserved keyword.

```
<?php
$var = 'Bob';
$Var = 'Joe';
$VAR = 'Mary';
echo "$var, $Var, $VAR";      // outputs "Bob, Joe, Mary"

$4site = 'not yet';     // invalid; starts with a number
```

```php
$_4site = 'not yet';    // valid; starts with an underscore
?>
```

3.1 How to generate variable names dynamically

A variable name can be set and used dynamically. This feature adds flexibility to our code when is properly used. Let's see an example:

```php
$a="Hello";
$$a=" world!";  // we are declaring the variable $Hello
echo "$a ${$a}"; // the output will be Hello world!
echo "$a $Hello"; // the output will be Hello world!
```

Dynamically generated variables can be useful for creating dynamic code, generate automatic forms, etc. We'll be using this feature in the future.

4 Constants

A **constant** is a identifier with a value that can't be changed while the application is running. In PHP constants names are usually written in uppercase and without the $ symbol. To declare a constant we must use the `define` function:

```php
define("PI", 3.14159);
echo PI; // 3.14159
```

> Constants are case-sensitive. By convention, the names of constants are written in uppercase.
>
> The constants **`true`**, **`false`** and **`null`** are case_insensitive. Exemple: `True`, `TRUE`, `tRuE`, etc. are valid.

PHP language provides some **predefined constants**. Many of them only are available when using certain PHP extensions. Follow this link to get a list of the constants available as part of the PHP core (without extensions). `M_PI` is one of them, so we don't need to declare the `PI` constant like seen in the previous example.

We can use predefined constants to get information about the PHP environment, like the version of PHP we are using.

```php
echo PHP_VERSION; // 8.0.0
echo PHP_OS; // WINNT
```

Some of the predefined constants are called **magic constants** because their value can change depending on where they are used. For example, **__LINE__** returns the current line number of the file. More magic constants following this link.

Magic constants are useful for debugging and logging.

```php
echo __LINE__; // 3
echo __FILE__; // C:\xampp\htdocs\curs2425\ud02-01.php
echo __DIR__; // C:\xampp\htdocs\curs2425
```

Some other **Magic constants** are:

- `__FUNCTION__`: The function name (only if used inside a function)
- `__CLASS__`: The class name (only if used inside a class)
- `__METHOD__`: The class method name (only if used inside a class)
- `__NAMESPACE__`: The name of the current namespace (only if used inside a namespace)
- `__TRAIT__`: The trait name (only if used inside a trait)

5 Operators

Operators are used to perform operations on variables and values. PHP supports a wide range of operators, including arithmetic, comparison, assignment, logical, and string operators.

5.1 Arithmetic operators

The arithmetic operators include the four basic arithmetic operations (addition, substraction, multiplication and division), as well as the modulus operator (%), which is used to obtain the division remainder, and the exponentiation operator (**, from PHP 5.6), that can be replaced by the function pow().

```php
$x = 4 + 2; // 6 // addition
$x = 4 - 2; // 2 // subtraction
$x = 4 * 2; // 8 // multiplication
$x = 4 / 2; // 2 // division
$x = 4 % 2; // 0 // modulus (division remainder)
```

```php
$x = 4 ** 2; // 16 // exponentiation, equals to $x=pow(4,2);
$x = -$x; //Negation, an inversion of the sign, so positive numbers become
↪  negative and negative numbers become positive.
```

5.2 Comparison operators

The comparison operators compare two values and return either **true** or **false**. They are mainly used to specify conditions, which are expressions that evaluate to either true or false.

```php
$x = (2 == 3); // false // equal to
$x = (2 != 3); // true // not equal to
$x = (2 <> 3); // true // not equal to (alternative)
$x = (2 === 3); // false // identical (type and content)
$x = (2 !== 3); // true // not identical
$x = (2 > 3); // false // greater than
$x = (2 < 3); // true // less than
$x = (2 >= 3); // false // greater than or equal to
$x = (2 <= 3); // true // less than or equal to
```

> The strict equality operators, `===` and `!==`, are used for comparing both type and value.

```php
$x = (2 == 2); // true
$x = (2 == '2'); // true
$x = (2 === 2); // true
$x = (2 === '2'); // false
```

5.3 Assignment and combined assignment operators

The basic assignment operator is `"="`. It means that the left operand gets set to the value of the expression on the right.

```php
$x = 5; // the value of $x is 5
```

A common use of the assignment and arithmetic operators is to operate on a variable and then save the result back into that same variable. These operations can be shortened with the **combined assignment operators**.

```php
$x += 5;    // equals to $x = $x+5;
$x -= 5;    // equals to $x = $x-5;
$x *= 5;    // equals to $x = $x*5;
$x /= 5;    // equals to $x = $x/5;
$x %= 5;    // equals to $x = $x%5;
$x **= 5;   // equals to $x = $x**5;
```

5.4 Increment and decrement operators

Another common operation is to increment or decrement a variable by one. This can be simplified with the increment (++) and decrement (−−) operators.

```php
$x++; // equals to $x += 1;
$x--; // equals to $x -= 1;
```

5.5 String concatenation

PHP has two string operators. The dot symbol is known as the ***concatenation operator*** (`.`). It combines two strings into one.

```php
$a = "Hello";
$b = $a . " World"; // the value of $b is Hello World
```

A numeric variable can be concatenated with a string.

```php
$a = 1;
$b = 2;
echo "a variable is ".$a." and b variable is ".$b;
// Result: a variable is 1 and b variable is 2
```

Combined assignment operator can be used in this case too.

```php
$a = "Hello";
$a. = " world"; // the value of $a is "Hello world"
```

5.6 Logical operators

Logical operators are often used together with the comparison operators.

- **and - &&**: logical and (&&) evaluates to true if both the left and right side of the expression are true
- **or - ||**: logical or (||) evaluates to true if either the left or right side is true. And, of course, if both are true.
- **!: Logical not**: inverts the logical value of the expression from true to false, or from false to true

```php
$x = (true && false);   // Value of $x: false // logical and
$x = (true and false);  // Value of $x: false // logical and
$x = (true || false);   // Value of $x: true  // logical or
$x = (true or false);   // Value of $x: true  // logical or
$x = !(true);           // Value of $x: false // logical not
$y = true;
$x = !$y;               // Value of $x: false // logical not
```

6 Basic data types

PHP, as well as many other interpreted languages, is a loosely-typed programming language. That means that we can create variables by simply assigning a value without explicitly declaring a type. The type of the variable is determined by the value assigned to it.

```php
$variable="Hello, world!"; // string
$variable=10; // integer
$variable=10.5; // float
$variable=true; // boolean
```

However, since the version 7.0, PHP allows us to declare the type of a variable inside functions, classes or methods. This is called **type hinting**. PHP includes some data types that can be used to declare the types in these situations.

Let's take a look to some of most used scalar data types:

- **int** / **integer** (integer numbers)
- **float** / **double** (real numbers)
- **string** (strings)

- **bool** (boolean)
- **null** (null value)

```php
function sum(int $a, int $b): int {  // the function can only receive integers
 ↪  and return an integer
    return $a + $b;
}
```

From 7.4. we can use the **`mixed`** type, which can be any type. This is useful when a function can accept multiple data types or when the exact type cannot be determined beforehand.

```php
function sum(mixed $a, mixed $b): mixed {
    return $a + $b;
}
// the function can be used to sum two integers, two floats or a float and
 ↪  an integer
```

The operator ? before a type means that the variable can be of that type or null.

```php
function operation(?int $a, ?int $b): ?int {
    return...;
}
// the function can be used to operate with two integers or two null values
// and can return an integer or null
```

We can use the operator **|** to indicate that a variable can be of one type or another.

```php
function operation(int | float $a, int | float $b): int | float {
    return...;
}
// the function can be used to operate with two integers, two floats or an
 ↪  integer and a float, and can return an integer or a float
```

The **`void`** type means that the function doesn't return anything.

```php
function sayHello(): void {
    echo "Hello, world!";
}
```

To ask for the type of a variable or expression, we can use the **gettype()** function. Let's see some examples:

```php
$variable1="Hello, world!";
$variable2="10";
$variable3=10;
$variable4=10.5;
echo "Type of variable1: ".gettype($variable1)."<br/>";
echo "Type of variable2: ".gettype($variable2)."<br/>";
echo "Type of variable3: ".gettype($variable3)."<br/>";
echo "Type of variable4: ".gettype($variable4)."<br/>";
```

The output of the previous code:

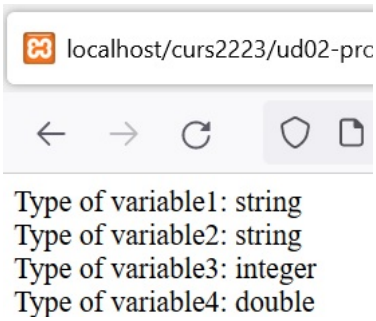

Figure 5: Data types

We can ask if a variable belongs to a certain type by using functions like:

- **is_bool($var)**: returns true if the variable is boolean
- **is_float($var)**: returns true if the variable is float
- **is_integer($var)**: returns true if the variable is integer
- **is_numeric($var)**: returns true if the variable is numeric
- **is_string($var)**: returns true if the variable is string
- **is_null($var)**: returns true if the variable is null

> In case we have a declaration like `$variable="100"`, both functions `is_numeric($variable)` and `is_string($variable)` will return true. However, the function `is_integer($variable)` will return false.

While debugging we can use the `var_dump()` function to show the type and value of a variable. This function should be deleted in production. It's easier to search and delete `var_dump` than `echo` or `print`.

```php
$a = 32;
echo var_dump($a) . "<br>"; //prints int(32)

$b = "Hello world!";
echo var_dump($b) . "<br>"; //prints string(12) "Hello world!"

$c = 32.5;
echo var_dump($c) . "<br>"; //prints float(32.5)

// Dump two variables
echo var_dump($a, $b) . "<br>"; //prints int(32) string(12) "Hello world!"
?>
```

7 PHP and HTML

As we have seen in the previous examples, we can merge PHP code and HTML tags. We can do it in two different ways.

PHP code inside HTML tags

```php
<h2>This line shows the value of a PHP expression <?php echo $expression;
↪  ?> </h2>
// or, using the short echo tag,
<h2>This line shows the value of a PHP expression <?= $expression ?> </h2>
```

HTML tags inside PHP code

```php
<?php
echo "<h2> This line shows the value of a PHP expression $expression
↪  </h2>";
?>
```

We will decide which one works best in every single situation.

> **Important:** the `echo` statement doesn't add a line break unless we include tags as HTML `<br>`, headings (`<h1>...<h6>`) or similar.

```php
echo "Hello, world!"; // doesn't jumps to the next line
echo "Hello, world!<br>"; // jumps to the next line
echo "<h3>Hello, world!</h3>"; // jumps to the next line
```

Sometimes, when the PHP code is a little more complex, we will need to concatenate the HTML and PHP sections of the expression.

```php
define("PI",3.14159);
$radius=5;
echo "The length of the circumference with radius $radius is
 ↪  ".(2*PI*$radius);
```

Figure 6: Length of the circumference

7.1 `echo` and `print` statements

In PHP we have two ways to get output: `echo` and `print`. They are more or less the same. They are both used to output data to the screen.

The differences between `echo` and `print` are:

- `echo` has no return value while `print` has a return value of 1 so it can be used in expressions
- `echo` can take multiple parameters (although such usage is rare) while `print` can take one argument
- `echo` is a little bit faster than `print`

Both can be used with or without parentheses: `echo` or `echo()`, `print` or `print()`.

Examples:

```php
<?php
echo "<h2>PHP is Fun!</h2>";
echo("Hello world!<br>");
echo "This ", "string ", "was ", "made ", "with multiple parameters.";
?>
```

```php
<?php
$value = print("Hello world!<br>"); // This will output "Hello world!" and
↪  return 1
print "<p> Value returned by print(): $value </p>"; // Value returned by
↪  print(): 1
print "This ", "will ", "fail "; // Error
?>
```

8 Query strings

A query string is a part of a uniform resource locator (URL) that assigns values to specified parameters.

Typical URL containing a query string is as follows:

`https://example.com/script.php?name=Jane`

The question mark **?** is used as a separator, and is not part of the query string. It marks the beginning of the query string.

If we want to pass multiple params, we use the ampersand **&**:

`https://example.com/script.php?name=Jane&surname=Doe`

We can read the values of the query string in a PHP script via the predefined array `$_GET`:

```php
$name = $_GET["name"];
$surname = $_GET["surname"];
```

This is a simple way to pass parameters to PHP scripts before we learn how to do it via forms. Create in your `provesPHP` folder a file named `proves.php` with the following content:

```php
<?php
$name = $_GET["name"];
$surname = $_GET["surname"];
echo "Hello, $name $surname!";
```

Call the previous script from the URL by typing in your browser:

`http://localhost/provesPHP/proves.php?name=YourName&surname=Yoursurname`

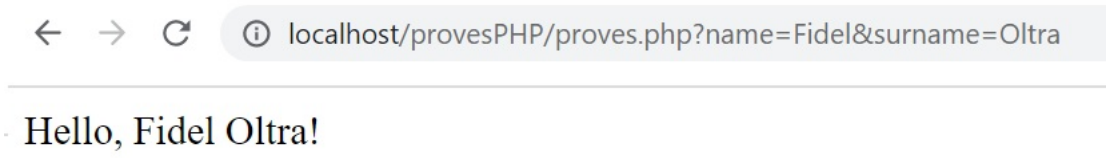

Figure 7: Script with query strings

9 Strings

We don't need to declare something as a "string" in PHP. Any variable which content is enclosed in double or single quotes is a string.

```
$variable="Hello";
$variable='Hello';
```

If a string is delimited by double quotes, there can be any number of single quotes inside it, and vice versa.

Just remember that variables in double quotes are changed by its value, but variables in single quotes don't.

```
<?php
$variable = "world";
echo "<p>Hello $variable</p>"; //Outputs "Hello world"
echo '<p>Hello $variable</p>'; //Outputs "Hello $variable"
?>
```

The only escape sequences that work in **single quotes** are **\'** (for escape a single quote) and **** (for escape a backslash).

There are many escape sequences for **double quotes**:

Sequence	Meaning
\n	line break
\t	horizontal tab
\\	backslash
\$	dollar sign

Sequence	Meaning
\"	double-quote

If we want that the generated HTML page had a readable format, with indentations and line breaks, we must use escape sequences such as \n.

For example, the next PHP code generates the HTML code below:

```php
<?php
echo "<ul>";
echo "<li>One</li>";
echo "<li>Two</li>";
echo "</ul>\n";
?>
```

```
<ul><li>One</li><li>Two</li></ul>
```

The next two PHP codes generate a well formatted HTML code:

```php
<?php
echo "<ul>
  <li>One</li>
  <li>Two</li>
</ul>\n";
?>
```

```php
<?php
echo "<ul>\n  <li>One</li>\n  <li>Two</li>\n</ul>\n";
?>
```

```
<ul>
  <li>One</li>
  <li>Two</li>
</ul>
```

As we have already seen, we can print strings using the instruction **`echo`** or **`print`**. With **echo** we can combine variables and text using commas (`,`). But with **print** we can't.

```php
$name="David";
echo "Hello, ", $name;
// or
echo "Hello, ".$name;
// or
print "Hello, ".$name; //Ok

print "Hello, ", $name; //Error
```

When the name of a variable inside a string is followed by characters that can be part of the name, the PHP interpreter takes these characters as part of the variable name:

```php
<?php
$fontSize = 40;
echo "<p style='font-size: $fontSizepx;'> Parsing variables test</p>";
//Doesn't work because the variable $fontSizepx is not defined
?>
```

We need to wrap the variable's name with curly brackets **{}** in order to parse the variable correctly:

```php
<?php
$fontSize = 40;
echo "<p style='font-size: {$fontSize}px;'> Parsing variables test</p>";
//It works!
?>
```

> We can wrap the whole variable including the $, like in `{$fontSize}`, or only the variable's name, like in `${fontSize}`.

9.1 heredoc and nowdoc

Another way to create long strings is by using **heredoc** and **nowdoc** notations.

Heredoc syntax consists of the <<< operator followed by an identifier and a new line. The string is then included followed by a new line containing the identifier to close the string. **Variables are parsed** inside of a heredoc string, just as with double-quoted strings.

```
$variable = <<<LABEL
Heredoc (with parsing)
LABEL;
```

Nowdoc syntax is very similar, but **the initial identifier is enclosed in single quotes** instead of double quotes. **Variables are not parsed** inside a nowdoc string.

```
$variable = <<<'LABEL'
Nowdoc (without parsing)
LABEL;
```

Let's see an example:

```
<?php
$name="David";

$variable1 = <<<LABEL
Heredoc (with parsing)
This $name will be parsed
LABEL;

$variable2 = <<<'LABEL'
Nowdoc (without parsing)
This $name won't be parsed
LABEL;

echo "<h2>With Heredoc</h2>";
echo $variable1;
echo "<h2>With Nowdoc</h2>";
echo $variable2;
?>
```

The result, as shown on screen, is:

nowdoc is ideal for embedding PHP code or other large blocks of text without the need for escaping. It's similar to the XML **<![CDATA[ ]]>** construct, in that it declares a block of text which is not for parsing.

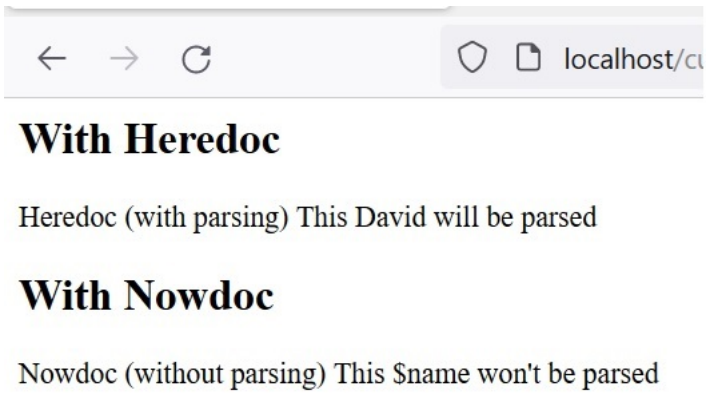

Figure 8: Strings with Heredoc and Nowdoc

9.2 printf and sprintf

The functions **`printf`** and **`sprintf`** can be used with strings to output a formatted string to the screen (`printf`) or to another variable (`sprintf`).

```php
$name="John";
$age=33;

// with echo
echo "Hello, $name, you are $age years old <br/> ";

// with printf
printf("Hello, %s, you are %d years old<br/>", $name, $age);

// with sprintf
$newString=sprintf("Hello, %s, you are %d years old<br/>", $name, $age);

echo $newString;
```

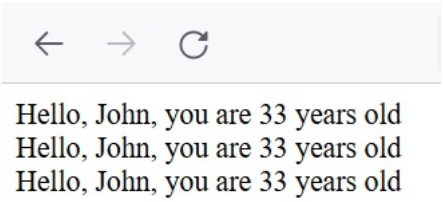

Figure 9: Strings with echo, printf and sprintf

The result is the same in each case.

A conversion specification follows this prototype:

`%[argnum$][flags][width][.precision]specifier`

The most common format values are:

- **%%** - Returns a percent sign
- **%b** - Binary number
- **%c** - The character according to the ASCII value
- **%d** - Signed decimal number (negative, zero or positive)
- **%e** - Scientific notation using a lowercase (e.g. 1.2e+2)
- **%u** - Unsigned decimal number (equal to or greater than zero)
- **%f** - Floating-point number (local settings aware)
- **%F** - Floating-point number (not local settings aware)
- **%s** - String
- **%x** - Hexadecimal number (lowercase letters)

An example:

```php
$number=10;
printf("The binary representation of %d is %b", $number, $number);
// The binary representation of 10 is 1010
```

More about sprintf and printf formats: https://www.php.net/manual/en/function.sprintf.php

> The **nl2br()** function can be used to convert special newline characters (\n) to actual newlines on the system we are working on (
 label in HTML). An exception happens if the text is enclosed in single quotes because the \n will not be parsed.

```php
<?php
$text="Hello\nworld!";
echo "Without nl2br():<br>";
echo $text;
echo "<hr>";
echo "With nl2br():<br>";
echo nl2br($text);
?>
```

9.3 String functions

In PHP we have a lot of functions to manipulate strings. Some of them are shown in the next table.

Figure 10: Same text with and without nl2br()

Function	Utility	Example
substr	Returns a section of a string	`$sub=substr($mainString,initPos,length)`
substr_replace	Replace text within a string	`$newString=substr_replace($mainString,$new init, length);`
str_replace	Replace all occurrences of the search string in the main string with the replace string	`$new=str_replace($search,$replace,$mainSt`
strlen	Returns the length of a string	`$len=strlen($string);`
strpos	Finds the position of the first occurrence of a substring in a string. Since the first position of a string is 0, strpos returns FALSE if the substring is not found	`$pos=strpos($mainString,$subString);`
strrpos	Finds the position of the last occurrence of a substring in a string (returns FALSE if not found)	`$lastPos=strrpos($mainString,$subString);`
ltrim	Strip left whitespaces from a string	`$newString=ltrim($oldString);`
rtrim	Strip right whitespaces from a string	`$newString=rtrim($oldString);`
trim	Strip both left and right whitespaces from a string	`$newString=trim($oldString);`
strtolower	Converts a string to lowercase	`$lowString=strtolower($mainString);`

Function	Utility	Example
strtoupper	Converts a string to uppercase	`$upperString=strtoupper($mainString);`
strchr / **strstr**	Finds the first occurrence of a substring inside a string and returns the left or right (depends on the third parameter of the function) part of the main string from the substring position (or FALSE if not found)	`$returnedString=strchr($mainString,$searc`
strrchr	Finds the last ocurrence of a character on a string and returns the portion from the position where the character has been found until the end of the string (or FALSE if not found)	`$returnedString=strrchr($mainString,$char`
strrev	Returns a string reversed	`$reversedString=strrev($originalString);`

Some other string functions are:

- **str_word_count()**: Counts the number of words in a string.
- **str_repeat()**: Repeats a string a specified number of times.

For a more detailed explanation of these functions and for a lot of other string functions, follow this link:

String functions in PHP

10 Dates and time

Dates are a very common data type used in applications and databases. For that reason PHP has a lot of functions to handle dates. To avoid confusion, we will use the **DateTime class** to work with dates.

Before entering into the DateTime class, let's see some functions to get the current date and time.

The **`date()`** function is used to format a Unix timestamp according to the given format. The first parameter is the format and the second one is the timestamp. If the second parameter is omitted, the current date and time will be used.

```php
// Returns a formatted date string
date(string $format, ?int $timestamp = null): string
```

Example:

```php
echo "Today is ".date("l, d-m-Y");
// Today is Monday, 29-08-2022
```

- d: month day
- m: month (numeric)
- Y: year (4 digits)
- l: day of week (text)

More date formats in the next link: Date formats

10.1 The DateTime class

The **DateTime class** is a built-in class in PHP that allows us to work with dates and times. It provides a lot of methods to manipulate dates and times.

A **DateTime object** is an instance of the **DateTime class** which represents a date and time. We can create a new **DateTime** object using the **new** keyword followed by the **DateTime class** name.

The constructor of **DateTime** accepts a string parameter which defaults to "now", the current time and date. To create an object for a specific date, you should pass the specific date and time to it.

```php
<?php
$today = new DateTime(); // Objeto DateTime with current date/time
$tomorrow= new DateTime('tomorrow'); // Objeto DateTime with tomorrow date
$theDayAfterTomorrow = new DateTime('+2 days'); //DateTime representing 2
↪  days from now on.
```

```php
$day1 = new DateTime('2024, September 13'); //DateTime representing 2024,
↪  September 13
$day2 = new DateTime('2024-09-13'); //DateTime representing 2024, September
↪  13
?>
```

To get the current date and time, we can use the **`format()`** method. This method accepts a string parameter with the format we want to use to display the date and time.

```php
echo $today->format('l, d-m-Y H:i:s'); // Friday 13-09-2024 16:19:33
// or
echo date_format($today, 'l, d-m-Y H:i:s'); // Friday 13-09-2024 16:19:33
```

We can create a date object with any other date and format using the **date_create_from_format()** function.

```php
// Returns a new DateTime object representing the date and time specified
↪  by the datetime string, which was formatted in the given format.
date_create_from_format(string $format, string $datetime, ?DateTimeZone
↪  $timezone = null): DateTime
```

Example:

```php
$newDate = date_create_from_format('d-M-Y', '12-Sep-2024');
echo date_format($newDate, 'l, d-m-Y'); // Thursday, 12-09-2024
```

We can create a DateTime object using a timestamp. A timestamp is a sequence of characters or encoded information identifying when a certain event occurred, usually giving date and time of day, sometimes accurate to a small fraction of a second.

```php
$timestamp = 1631520000; // 13-09-2021 00:00:00
$date = new DateTime();
$date->setTimestamp($timestamp);
echo $date->format('l, d-m-Y H:i:s'); // Monday, 13-09-2021 00:00:00
```

We can also use the **`mktime()`** function to create a date. It creates a **timestamp**.

```php
// Returns the Unix timestamp (long integer) corresponding to the arguments
↪  given.
mktime(
    int $hour,
    ?int $minute = null,
    ?int $second = null,
    ?int $month = null,
    ?int $day = null,
    ?int $year = null
): int
```

Example:

```php
$date=mktime(0,0,0,09,10,2024);
// hours, minutes, seconds, month, day, year
echo date("l, d-m-Y H:i:s",$date); // Tuesday, 10-09-2024 00:00:00
$day04=new DateTime();
$day04->setTimestamp(mktime(0,0,0,9,10,2024));
echo $day04->format('l, d-m-Y H:i:s'); // Tuesday, 10-09-2024 00:00:00
```

Once the date is created with **mktime()**, the function **getdate()** can be used to extract some information. This function returns an **associative array** with these indexes: ***seconds***, ***minutes***, ***hours***, ***mday*** (day of month), ***wday*** (day of week in number), ***mon*** (month in number), ***year***, ***weekday*** (day of week in letters) and ***month*** (month in letters).

```php
// Returns an associative array containing the date information of the
↪  timestamp, or the current local time if timestamp is omitted or null.
getdate(?int $timestamp = null): array
```

Example:

```php
$date=mktime(21,58,40,8,31,2022);
$info=getdate($date);
echo $info["year"]; // Output: 2022
```

Some methods we can use with **DateTime objects** are:

- **add()**: Adds an amount of days, months, years, hours, minutes and seconds to a DateTime object.

- **`sub()`**: Subtracts an amount of days, months, years, hours, minutes and seconds from a DateTime object.
- **`diff()`**: Returns the difference between two DateTime objects.
- **`format()`**: Returns date formatted according to given format.
- **`modify()`**: Alters the timestamp of a DateTime object.
- **`setDate()`**: Sets the date.
- **`setTime()`**: Sets the time.
- **`setTimestamp()`**: Sets the date and time based on a Unix timestamp.
- **`getTimestamp()`**: Gets the Unix timestamp.

Example:

```php
$today = new DateTime();
$today->add(new DateInterval('P10D')); // adds 10 days
echo $today->format('l, d-m-Y H:i:s'); // Monday, 23-09-2024 16:19:33
```

> PHP also has the **`DateTimeImmutable`** class, that behaves the same as DateTime except new objects are returned when modification methods such as `DateTime::modify()` are called. Both classes inherited from **`DateTimeInterface`** interface.

To extract a single information of a DateTime object, we can use the **`format()`** method.

```php
$today = new DateTime();
$dayOfTheWeek = $today->format('l');
echo "Today is ".$dayOfTheWeek;  // Today is Friday
```

In order to calculate differences between dates, we can use the **`date_diff()`** function. This function returns a **DateInterval object**.

An example:

```php
$todayDate=new DateTime();
$inTenDays = new DateTime('+10 days');
$difference=date_diff($todayDate,$inTenDays);
// or $difference = $todayDate->diff($inTenDays);
$differenceInDays=$difference->format("%a");
echo "<h3>$differenceInDays days left to
↪  ".$inTenDays->format("d/m/Y")."</h3>";
// 10 days left to 23/09/2024
```

We can also work with differences between dates with the **`diff()`** method:

```php
$mary = new DateTime('May 20th, 1980');
$john = new DateTime('March 11th, 1962');

$diff = mary->diff(john);
var_dump($diff);
//Outputs: DateInterval Object ( [y] => 18 [m] => 2 [d] => 9 [h] => 0 [i]
↪  => 0 [s] => 0 [f] => 0 [invert] => 1 [days] => 6645 [from_string] => )
```

The `diff` method returns a **DateInterval** object. You can generate a friendly output from a DateInterval object:

```php
$diff = $mary->diff($john);
echo $diff->format('John is older by %Y years and %m months');
//Outputs: John is older by 18 years and 2 months
```

A very interesting thing is that you can apply a interval to another DateTime object:

```php
$john->sub($diff); //John's birthday changed to Mary's
echo $john->format('d-m-Y'); //20-05-1980
```

And you can create a new DateInterval form the constructor (see the constructor's documentation):

```php
$new_diff = new DateInterval('P2Y');
```

More about date and time features and functions in this link.

More about the DateTime class.

11 Arrays

An **array** is a collection of values stored in a single variable. Arrays in PHP consist of key-value pairs. The **key** (or **index**) can either be an integer (***numeric array***), a string (***associative array***), or a combination of both (**mixed array**). The value can be of any data type.

In the PHP language, arrays are **dynamic** because their size can change during the execution of the script. In addition, an array in PHP can contain different data types.

11.1 Numeric arrays

Arrays with numeric index or **Numeric arrays** store each element in the array with a numeric index. An array is created using the array constructor. This constructor takes a list of values, which are assigned to elements of the array.

```php
$arrayOne = array(1,2,3);
```

In this filled numeric arrays, **the first position of the array is 0**. Thus, in this case the element 1 is stored in the position 0, the element 2 is stored in the position 1, and the element 3 is stored in the position 2.

As of PHP 5.4, a shorter syntax is available, where the array constructor is replaced with square brackets.

```php
$arrayTwo = [1,2,3];
```

We can declare an array without a predefined content and fill it later. If we don't indicate a position, the new item will be added at the end of the array.

```php
$arrayThree = array(); // creates an empty array
$arrayThree[] = 1; // as the array is empty, this value will be stored in
↪  position 0
$arrayThree[] = 2; // stored in position 1
$arrayThree[] = 3; // stored in position 2
```

Once the array is created, its elements can be referenced by placing the index of the desired element in square brackets. Remember that the index begins with zero.

```php
echo $arrayThree[1]; // shows a 2
```

Positions and values can be added at any time.

```php
$arrayThree[5] = 4; // stored in position 5 even if the last position is
↪  currently 2
```

The function **`count()`** can be used to find out the number of elements of a given array. It works in numeric and no numeric arrays.

```php
echo count($arrayThree); // shows a 4 (positions 0, 1, 2 and 5)
```

The functions **`print_r()`** and **`var_dump()`** show the content of the array.

```php
$arrayThree = array();
$arrayThree[] = 1; // as the array is empty, this value will be stored in
↪  position 0
$arrayThree[] = 2; // stored in position 1
$arrayThree[] = 3; // stored in position 2
$arrayThree[5] = 4; // stored in position 5

echo "<h2>With print_r()</h2>";
print_r($arrayThree);
echo "<h2>With var_dump()</h2>";
var_dump($arrayThree);
```

11.2 Associative arrays

In associative arrays, **the key can be an integer or a string**. When creating the array the **double arrow operator (=>)** is used to tell which key refers to what value. Elements in associative arrays are referenced using the element keys as in numeric arrays.

```php
//Associative array with strings as keys
$phones=array("John"=>"911111111",
              "Carl"=>"912222222");
echo $phones["John"]; // 911111111
echo $phones["Carl"]; // 912222222
```

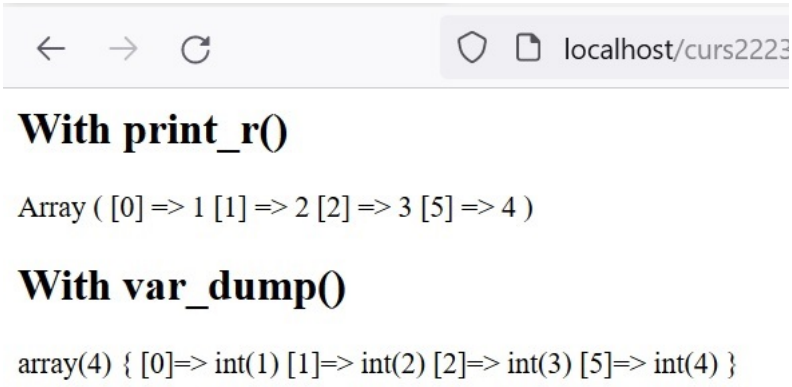

Figure 11: An array as shows with print_r and var_dump

```php
//Associative array with ints as keys
$squares = [3 => 9, 5 => 25, 10 => 100];
print "<p>The square of 3 is $squares[3]</p>\n"; //The square fo 3 is 9
```

When we add an element to an associative array without a specified position, the new item will be added at the end of the array. The key will follow the order of the positions defined as numeric, beginning with 0 if no numeric position has been previously defined.

```php
$phones=array("John"=>"911111111",
              "Carl"=>"912222222");
$phones[]="913333333"; // position 0
echo $phones["John"];  // 911111111
echo $phones["Carl"];  // 912222222
echo $phones[0];       // 913333333
```

11.3 Arrays operations

The operators **+** and **+=** can be used to join two arrays. The union of two arrays contains all the items of the first array and the items of the second array whose index is not in the first array:

```php
<?php
$x = array("red", "green");
$y = array("brown", "blue", "yellow");
```

```php
print_r($x + $y); // Array ( [0] => red [1] => green [2] => yellow )
?>
```

```php
<?php
$x = array("a" => "red", "b" => "green");
$y = array("b" => "brown", "c" =>"blue", "d" => "yellow");

print_r($x + $y); // Array ( [a] => red [b] => green [c] => blue [d] =>
↪  yellow )
?>
```

We can copy the content of one array into a new variable using the **=** operator. The two arrays are independent.

```php
<?php
$x = array("red", "green");
$y = $x;
print_r($y); // Array ( [0] => red [1] => green )
?>
```

We can use the **comparison operators** with arrays:

Operator	Name	Example	Result
==	Equality	`$x==$y`	Returns `true` if `$x` and `$y` have the same key/value pairs
===	Identity	`$x===$y`	Returns `true` if `$x` and `$y` have the same key/value pairs in the same order and of the same types
!=	Inequality	`$x!=$y`	Returns `true` if `$x` is not equal to `$y`
<>	Inequality	`$x<>$y`	Returns `true` if `$x` is not equal to `$y`
!==	Non-identity	`$x!==$y`	Returns `true` if `$x` is not identical to `$y`

11.4 Some other functions on arrays

Function	Utility	Example
unset()	Deletes an element of the array or even the whole array	`unset($array[$position])` `unset($array)`
array_values()	Returns a numeric array with the values of other array	`$newArray=array_values($oldArray)`
array_diff()	Returns and array with the differences between two arrays	`$diffArray=array_diff($array1,$array2)`
array_fill()	Fills a whole array with a single value	`$array=array_fill($init,$positions,$val`
array_search()	Checks if a value exists in the array	`$key=array_search($value,$array)`
array_key_exists()	Checks if a key exists in the array	`if (array_key_exists($key,$array))`
sort()	Sorts an indexed (numeric) array using the values in ascending order without keeping the association key->value	`sort($array)`
rsort()	Sorts an indexed (numeric) array using the values in descending order without keeping the association key->value	`rsort($array)`
asort()	Sorts an associative array using the values in ascending order and keeping the association key->value	`asort($array)`
arsort()	Sorts an associative array using the values in descending order and keeping the association key->value	`arsort($array)`
ksort()	Sorts an array using the keys in ascending order	`ksort($array)`

Function	Utility	Example
krsort()	Sorts an array using the keys in descending order	`krsort($array)`
array_slice()	Returns a section from another array	`$newArray=array_slice($array,$init,$pos`

11.5 Implode and explode

The **implode()** function returns a string with the elements of a given array and the given string as a separator.

```
$string=implode($separator, $array);
```

The **explode()** function slices a string using the given separator string and puts the resulting elements into an array.

```
$array=explode($separator, $string);
```

Let's see an example:

```
// replacing a separator character in a string with blanks
$string="A:string:with:a:separator:character";
echo "<strong>The original string:</strong> $string <br/>";

$array=explode(":",$string);
echo "<strong>The array:</strong> ";
print_r($array);
echo "<br/>";

$newString=implode(" ",$array);
echo "<strong>The new string:</strong> $newString";
```

11.6 Multidimensional arrays

A multidimensional array is an array which elements can contain another array. A typical example is a matrix.

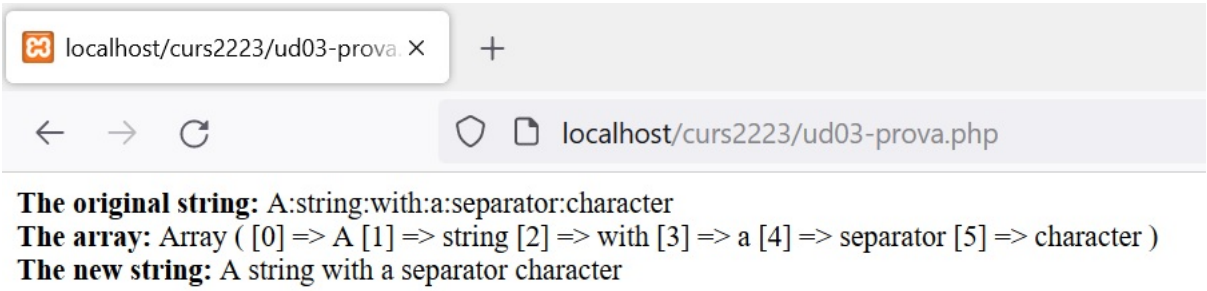

Figure 12: Implode and explode

1	2	3
4	5	6
7	8	9

Figure 13: A matrix (Two-dimensional array)

To create the two-dimensional array of the figure, we will create an array of 3 positions where each position contains an array with 3 elements.

```
$matrix=array(array(1,2,3),
              array(4,5,6),
              array(7,8,9));
```

The same technique works for associative arrays with multiple dimensions:

```
$friends=array("John"=>array("phone"=>"91111111","email"=>"john@gmail.com"),
    "Mary"=>array("phone"=>"92222222","Email"=>"mary@gmail.com"),
    "David"=>array("phone"=>"93333333","Email"=>"david@hotmail.com"));
```

In the next unit we will learn how to work with arrays using loops.

11.7 Arrays in strings

Indexed arrays can be inserted directly on strings:

```php
<?php
$cars = array("Volvo", "BMW", "Toyota");
echo "I like $cars[0], $cars[1] and $cars[2].";
?>
```

In the case of multidimensional arrays, we need to wrap the array with curly brackets:

```php
<?php
$name = "Jon Doe";
$greetings = [["Hello", "Hola"], ["Goodbye", "Adios"]];
print "<p> {$greetings[0][0]}, $name! How are you?</p>\n";
?>
```

And the same with associative arrays, using single quotes for the key's name if the string is inside double quotes:

```php
<?php
$age = array("Peter"=>"35", "Ben"=>"37", "Joe"=>"43");
echo "Peter is {$age['Peter']}  years old.";
?>
```

Remember that we can always use the concatenation operator.

```php
<?php
$age = array("Peter"=>"35", "Ben"=>"37", "Joe"=>"43");
echo "Peter is " . $age['Peter'] . " years old.";
?>
```

12 References

- PHP Manual
- W3Schools
- PHP notes for professionals (book)