420-DW3-AS Web Server Applications Development I

Winter 2023 – Week 2 : PHP Basics

PHP Basics

This week:

- Variables and constants,
- Types in PHP
- PHP Operators
- PHP Control statements

Code demonstrations on GitHub.

PHP Basics : Variables

NOTE: By variables, we mean here script-level variables, not OOP class fields/properties/attributes

In PHP, variables are ***named references*** to values in memory. Variable names must follow a few naming rules and have a few properties in and of themselves:

- Variables are represented by a dollar symbol (‘$’) followed by their name.
- Their name MUST start by a letter or an underscore (‘_’) followed by any number of letters, numbers and the underscore character (‘_’). **Variable names are case-sensitive**.
- Variables are not declared with a strict type, but they can possess a type based on the context.
- Declared but uninitialized variables start with a default value depending on their type, itself depending on the context.
- ***By default, variables are assigned by value,*** which means that the value assigned is copied and the variable it is assigned to points to the copy (special case for objects, see the type entry for objects in the following slides for explanations).
- Style conventions suggest using lowercase snake_case for their names.

Examples :

```
$my_variable;                   // $my_variable is NULL (with a warning/error about uninitialized or undefined variable)
$my_variable = 5;               // $my_variable is now an int(5)
$my_variable = “a string”;      // $my_variable is now a string(8)
$my_variable = true;            // $my_variable is now a bool(true)
$a_new_variable += 5;           // context is arithmetic addition, $a_new_variable will default to zero (int(0)) then be added 5 to its value.
```

PHP Basics : Constants

In PHP, constants behave similarly to variables as they are also ***named references*** to values in memory but with the following differences:

- They do NOT need a dollar sign as the first character of their representation
- They are declared using the const keyword or the define(string: $name, mixed: $value) built-in function. Note that style conventions prefer the const syntax in modern PHP.
- Once a value is assigned to them (including initialization), it cannot be rewritten, making them read-only.
- Style conventions suggest using UPPERCASE SNAKE_CASE for constant names.

Examples :

```
const MY_CONSTANT = 5;
const ANOTHER_CONSTANT = “a constant string”;
define(“A_THIRD_CONSTANT”, true);

if (A_THIRD_CONSTANT) {
        $a_variable = ANOTHER_CONSTANT;
}
```

PHP Basics : Types

Although PHP is not in and of itself a strong-typed language, it nonetheless possess types. If types cannot be declared for variables, they can be for function parameters and function return value, as well as for OOP structures. PHP does a lot of implicit type conversions depending on the context; **for further information, see** PHP Type Juggling.

- **NULL**: the simple no-value type in PHP. Undefined and un-set variables default to to its unique value, null. The literal value null is case-insensitive so null is equivalent to NULL.
- **Bool**: the bool type can only take two values: true or false. Both literal values are case-insensitive like null. Other values can be implicitly converted to bool when used in a logical context:
  - When converting to bool, the following values are considered false:
    - the boolean false itself
    - the int 0 (zero) (but not negative numbers, which are considered true)
    - the floats 0.0 and -0.0 (zero) (but not other negative numbers, which are considered true)
    - the empty string "", and the string "0"
    - an array with zero elements
    - the unitary type NULL (including unset variables)
    - Internal objects that overload their casting behaviour to bool. For example: SimpleXML objects created from empty elements without attributes.
  - Every other value is considered true (including resource and NAN).

PHP Basics : Types (continued)

- **Integers** (int): normal integer values. PHP does not make a difference between short and long integers; ***the maximum and minimum values acceptable by an int depends on the platform PHP is executing on*** and can be checked by using the built-in PHP_INT_SIZE, PHP_INT_MAX and PHP_INT_MIN constants. PHP supports binary, octal, decimal and hexadecimal literal values for integers; see PHP integers for details.
- **Floating-point numbers** (float, also named “double”): normal floating-point number values. Like for integers, PHP does not have short and long floating point numbers. NOTE: like most programming languages, PHP floats ***are vulnerable to the floating point precision problem of fixed-size binary representation***. PHP supports both standard and scientific notations for floating-point numbers; see PHP floats for details.
- **Strings** (string): normal strings similar to other programming languages. PHP supports single-quoted, double-quoted, HEREDOC syntax and NOWDOC syntax strings. PHP strings support escape sequences that use a backslash (‘\’) as the escape character (like ‘\n’ to insert a new line character); double-quoted and HEREDOC syntax strings support ***variable interpolation*** (e.g. writing a variable in one of those will have it replaced by its string-conversion value when the string is evaluated). See PHP strings for details. In many cases, PHP is able to interpret numeric strings as numbers without direct type cast (1 + “3” will result in 4). See PHP numeric strings for details.

PHP Basics : Types (continued)

- **Arrays** (array): arrays in PHP are actually ***ordered maps*** that associate ***keys*** to ***values***. They do not have a fixed length and can grow in size. Arrays can be of two types: indexed (with integer-only keys, starting at zero; the usual behavior of arrays in other languages) or associative (with string or mixed string/integer keys). All arrays are *de facto* associative, but when they have only integer-type keys, they are said to be indexed arrays, and search/find operations are optimized in those. Arrays are created with the array(<values>) language construct or with a shorthand (square brackets '[ ]') [<comma separated values>] syntax. Style conventions prefer the shorthand syntax in modern PHP.

- **Objects** (object or the class name): PHP allows its user to define their own types through Object-oriented programming. By defining a class, PHP developers create their own types of objects with specific *properties*, *class constants* and *methods*. PHP also supports the transformation of other types (notably arrays) to objects. The result is an object of type stdClass (built-in class). ***NOTE: The value of PHP object variables is actually a reference to the object in memory. Therefore, although they are still passed by value (passing the reference), they behave like if they were passed by reference (the references points to the same object in memory)***. Copying an object in PHP requires the use of the **clone** keyword. Finally, PHP objects are fully dynamic; this means that you can create properties in objects at run-time, properties that are not in the class definition. However, dynamic definition of object members is considered a bad practice and is strongly discouraged.

- **Enumerations** (enum or enumeration name): Enumerations are a special type of user-defined objects that only contain a strict set of values (*pure enums*) or a strict set of key-value pairs (*backed enums*).

- **Resource** (resource): Resource are a special type the represent references to external resources such as file handles, database connections etc… They are usually created and used by special functions. They cannot be converted to other types.

PHP Basics : Types (continued)

- **Callbacks/callables** (callable): It is possible to define functions and methods that receive an function as a parameter under the type named callable (often called a ***callback function***). PHP has some built-in functions that do this, such as the usort(array &$array, callable $callback) array-sorting function. The callback type represent any form of callable structure of which PHP supports quite a few. See PHP callable for details.

- **mixed** (PHP 8.0+): The mixed pseudo-type represents any other type. It is equivalent to the union type of object|resource|array|string|float|int|bool|null.

- **void**: The void pseudo-type is used to define functions that return nothing. It can only be used as a return type, not as a parameter or property type.

- **never** (PHP 8.1+): The never pseudo-type is used to define functions that never terminate normally (they must either call exit() or throw an exception at some point, or be infinite loops). It can only be used as a return type, not as a parameter or property type.

- The relative class types **parent**, **self** and **static**: These pseudo-types are used inside of a class definition to indicate types in relation to the class in which they are used. We define them more precisely when we get into the object-oriented PHP part of the course.

- **iterable**: The iterable pseudo-type is equivalent to the union type array|Traversable. It is used to indicate an object that can be ***traversed*** (using foreach, as an example)

PHP Basics : Including files

PHP being a script language, it can be useful to separate our code into multiple files unless we want to end up with one monstrously long file. However, because PHP responds to requests that target single files, we will need to tell PHP to specifically include the files we need into our main script. Including a file will simply tell PHP to evaluate its contents (execute it) and insert the contents at the inclusion point. You cannot use a variable, function or class defined in a file before that file is included. *In fact, you cannot use a variable, function or class before its definition has been evaluated by PHP; it needs to have been defined previously in the script (including a file that contains the definition does this) before it can be used*. This will become very important when we start using object-oriented PHP. Also, PHP does not allow you to define multiple functions with the same name or classes with the same fully qualified name. *Therefore you cannot include a file that defines functions more than once, an error will be thrown*.

There are a few directives that can be used to include files. The path is relative to the project root directory. Absolute paths are also OK:

- include “relative\path\to\file.php”;
- require “relative\path\to\file.php”; (like include, but throws fatal compilation errors if the file is not found)
- include_once “relative\path\to\file.php”; (includes the file only if it has not been included previously)
- require_once “relative\path\to\file.php”; (like include_once, but throws fatal compilation errors if the file is not found)

All of these directives can also be used with parentheses: include(“relative\path\to\file.php”)...

PHP Basics : Globals and superglobals

Since PHP executes files, both the main target of the web request and any included ones, the question arises as to what happens when we declare a variable in a file in respect to its existence in the others. This brings us to the concept of ***scopes***, or ***contexts***. Variables are defined in scopes, and exist only inside those. Scopes can be nested inside one another; what is defined in an outer scope is accessible inside an inner one. PHP introduces 3 different scope levels:

- Variables declared *outside* of functions belong to the ***global scope***, and are accessible from everywhere outside of function definitions.
- Variables declared *inside* of functions belong to a ***local function scope***, and are accessible only from within that function. They are created when the function is executed, and deleted when the function terminates. To access and use variables from the *global scope* from within a function definition, we must use the global keyword to "import" the global variable inside the function definition.
- Sometimes, we might want a variable defined in a local function scope to NOT be destroyed upon termination of its enclosing function. To achieve this, PHP possess the ***static scope***. Variables declared with the static keyword inside a function still exist only inside the function definition, but are not deleted after an execution terminates and remain available, with their value, in all executions of the function.

See PHP variable scopes for details.

PHP Basics : Globals and superglobals (continued)

PHP also introduces the concept of ***superglobals***: built-in associative array variables that are always accessible in all scopes; they do not need to use of the global keyword to be accessed, even inside function definitions.:

- **$GLOBALS**: an array of all the variables declared in the global scope
- **$_SERVER**: an array of various data about the server on which PHP is executing
- **$_GET**: an array of the parameters (query string) included in the received HTTP request
- **$_POST**: an array of the parameters included in the received HTTP request, if using the HTTP POST method.
- **$_FILES**: an array of items (including files) uploaded to the current script via the HTTP POST method.
- **$_COOKIE**: an array of variables passed to the current script via HTTP Cookies.
- **$_SESSION**: an array containing session variables available to the current script.
- **$_REQUEST**: an array that, by default, contains the contents of $_GET, $_POST and $_COOKIE.
- **$_ENV**: an array of variables passed to the current script via the environment method. Might be entirely disabled.

See PHP Superglobals for details.

PHP Basics : Operators

PHP Basics : Operators

List of PHP operators. See PHP Operators for details.

Arithmetic (basic math) operators:

- **Identity**: (‘+’ used as unary operator) Converts to int or float as appropriate. Example: +$number
- **Negation**: (‘-’ used as unary operator) Opposite of the operand. Example: -$number
- **Addition**: (‘+’) example: $number + 5
- **Subtraction**: (‘-’) example: $number – 5
- **Multiplication**: (‘*’) example: $number * 5
- **Division**: (‘/’) example: $number / 5
- **Modulo**: (‘%’) Remainder of integer division. Example: 7 % 4 results in 3 because 7/4 = 1 with 3 remaining
- **Exponentiation**: (‘**’): raises the left operand to the right operand’th power. Example: 2 ** 3 results in 8 (2^3=8)

PHP Basics : Operators (continued)

String operators :

- **Concatenation**: (‘.’) Appends the right operand to the end of the left operand. Example: “Hey” . “Jude” results in “HeyJude”.

Reference operator

- **Reference**: (‘&’) Returns the reference pointing to the value. Can only be used with named (identifiable) expressions like variables. Example: $same_value = &$value results in $same_value being an ***alias*** for $value; they refer to the same value in memory and thus that value can be changed through either variable.

Assignment operators:

- **Assignment**: (‘=’) Sets the the right expression operand as the value of the left operand. Example: $var = 64

Null-coalesce operator:

- **Null-coalesce**: (‘??’) Returns the left operand if it is **not** null. If it is, returns the right operand.

PHP Basics : Operators (continued)

Bitwise (bit operations in integers) operators :

- **Bitwise AND**: (‘&’) results in a new integer composed of the bits that are set in operand1 AND operand 2. Example (in binary integer literals): 0b11111000 & 0b00011111 results in 0b00011000
- **Bitwise OR**: (‘|’) results in a new integer composed of the bits that are set in operand1 OR operand2. Example: 0b11100011 | 0b00000111 results in 0b11100111
- **Bitwise XOR** (exclusive OR): (‘^’) results in a new integer composed of the bits that are set in operand 1 OR operand2 BUT NOT BOTH. Example: 0b00011110 | 0b00000111 results in 0b00011001
- **Bitwise NOT**: (‘~’ as unary operator) results in a new integer with the bits set and unset inverted. Example: ~0b11001100 results in 0b00110011
- **Bitwise shifts left and right** (‘<<’ and ‘>>’ respectively) Shifts the bits of the operand1 integer by operand2 steps to the right or left. Bits shifted off either end are discarded. Example 0b11111111 << 1 results in 0b11111110

PHP Basics : Operators (continued)

PHP also has ***compound operators***; operators that *do both a specific primary operation and an assignment operation* in one go. ***They require a variable as a left-side operand***. First, they do the primary operation for the left and right operands, and then assign the resulting value to the variable used as the left operand. Their operators are written by using the operator of the primary operation, followed by the assignment operator (ex,: ‘+=’, ‘.=’, ‘??=’ …).

The following primary operations have an associated compound operator:

- All non-unary arithmetic operations (+, -, *, /, %, **) → (+=, -=, *=, /=, %=, **=)
- The string concatenation operation (.) → (.=)
- All the bitwise operations (&, |, ^, <<, >>) → (&=, |=, ^=, <<=, >>=)
- The null-coalesce operation (??) → (??=)

PHP Basics : Operators (continued)

Comparison operators (usually evaluates to a boolean truth-value) :

- **Equality** (‘==’) true if the values of the operands are equal after type juggling. Examples: 4 == 5 is false, 4 == “4” is true.
- **Identical** (‘===’) true if the values of the operands are equal AND they are of the same type. Example: 4 === 4.0 is false.
- **Not-equal** (‘!=’ or ‘<>’) true if the values of the operands are not equal, after type juggling. Examples: 4 != 5 is true, 4 != “4” is false.
- **Not-identical** (‘!==’) true if the values of the operands are not equal OR if the operands are of different types. Example: 4 !== “4” is true.
- **Less than** and **Greater than** (‘<’ and ‘>’ respectively) true if the left operand is strictly less or strictly greater than the right operand, respectively. Examples: 4 < 5 is true, 4 < 4 is false, 8 > 5 is true, 8 > 8 is false.
- **Less or equal than** and **greater or equal than** (‘<=’ and ‘>=’ respectively) true if the left operand is less or equal, or greater or equal than, the right operand, respectively. Examples: 4 <= 4 is true, 4 <= 3 is false
- **Spaceship** (‘<=>’) returns an integer that is less than, equal to, or greater than zero when the left operand is less than, equal to, or greater than the right operand, respectively.

PHP Basics : Operators (continued)

Error control operator:

- **Suppression** (‘@’) Suppresses any diagnostic error that might be generated by the expression it is prepended to.

Execution operator:

- **Shell Execution** (‘` `’ backticks, not single-quotes) PHP will attempt to execute in the system shell (command line) whatever is written between the backticks and will return the output of such execution.

Incrementation/Decrementation operators (only affect numbers and strings):

- **Pre-incrementation** and **Pre-decrementation** (‘++’ and ‘--’ as unary operators respectively, placed before the operand) Increments or decements the operand by 1 respectively, then returns the incremented or decremented value. Examples: ++$number, --$number
- **Post-incrementation** and **Post-decrementation** (‘++’ and ‘--’ as unary operators respectively, placed after the operand) Returns the operand value THEN increments or decrements it respectively. Examples: $number++, $number--

PHP Basics : Operators (continued)

Logic operators:

- **AND** (‘&&’ or the and keyword) Evaluates to true if both operands themselves evaluate to true. Examples: (true && 1) is true, (true && false) is false, (true && 0) is false, (true && “”) is false, (true && null) is false
- **OR** (‘||’ or the or keyword) Evaluates to true if any of the operands evaluates to true. Starts the evaluation from the left, stops as soon as a true value is found. Example: (true || false || false || true) is true and stops any evaluation at the first true value.
- **NOT** (‘!’ as unary operator in front of the operand) Evaluates to true if the operand is ***NOT*** true and vice versa. Example: !true is false, !false is funny because it’s true.
- **XOR** (xor keyword) Evaluates to true if one **and only one** of the two operands is true (not both). Examples: (true xor false) is true, (true xor true) is **false**, (false xor false) is false.

Ternary operator:

- **Ternary operator** (three-operand operator: a ? b : c) Is equivalent to an if-else logic wrapped up in an expression: if leftmost operand is true, then returns the middle operand, otherwise the rightmost operand.

PHP Basics : Operators (continued)

Array operators (for arrays only):

- **Union** ('+') Returns the union array of both operand arrays, i.e., the right-hand array appended to the left-hand array; for keys that exist in both arrays, the elements from the left-hand array will be used, and the matching elements from the right-hand array will be ignored.
- **Equality** ('==') Evaluates to true if both arrays have the same key/value pairs.
- **Identity** ('===') Evaluates to true if both arrays have the same key/value pairs, in the same order and of the same types.
- **Inequality** ('!=' or '<>') Evaluates to true if the arrays are not equal (see Equality).
- **Non-identity** ('!==') Evaluates to true if the arrays are not identical (see Identity).
- **Array access pseudo-operator** ('[<key>]' as unary operator placed after the array operand) Returns the value associated with the key <key> in the array. If used with the assignment operator, can assign a value to the element in the array with key <key>. Can also be used with no given <key> value to add a value to an indexed array which will be inserted at its end. Examples: suppose that $array is an array; $array[0], $array["aKey"], $array["aKey"] = "potatoes", $array[] = 17

PHP Basics : Operators (continued)

Type operators:

- **Instanceof** (instanceof keyword) Evaluates to true if the left operand PHP variable is an instantiated object of a certain class, an instantiated object of a class that *inherits from* (***extends***) a certain class, or an instantiated object of a class that ***implements*** a certain interface. The right operand is the class or interface to test against. Example: ($object instanceof MyClass)

Callable operator:

- **Call pseudo-operator** (‘(<comma-separated arguments>)’ as unary operator placed after the callable operand) Executes (calls) the callable, passing it the list of arguments provided between the parentheses. It evaluates to the value returned by the call. Most, but not all, syntaxes of PHP callables support this operator. If you need to call one that does not support the pseudo-operator, see the call_user_func() built-in function. Example: suppose $my_function is a variable containing a callable; $my_function(“hello!”); calls the $my_function callable passing it the “hello!” string as argument.

PHP Basics : Control Structures

PHP Basics : if… elseif… else Statement

The if...else if… else… statement allows you to execute block(s) of statement(s) conditionally. The statements inside an if block will only be executed if the conditional expression specified evaluates to true

It is possible (not required) to add one or many other block(s) of statement(s) to be executed if the previous condition(s) are not met but if another specific condition is through the use of elseif subsequent block(s).

Finally, we can (not required) add a single block of statement(s) to be executed if none of the conditions specified by the previous if or elseif blocks are met through the use of an else block.

```
if (<first conditional>) {
   // statements to execute if the <first conditional> is true
} elseif (<second conditional>) {
   // statements to execute if the <first conditional> is false, but the <second conditional> is true.
} elseif (<third conditional>) {
   // statements to execute if the <first conditional> and the <second conditional> are false, but the <third conditional> is true.
} else {
   // statements to execute if none of the conditionals are true.
}
```

PHP Basics : switch… case Statement

The switch… case statement also allows you to execute block(s) of statement(s) conditionally, but does by checking if the evaluation of an expression (switch) is equal to specific values (cases). Each possible case is defined individually. The tests for each cases are sequential in a waterfall manner. Each case statement block must be terminated by a break instruction, otherwise the evaluation continues to the next cases. A special default case is declared is executed at the end unless the switch… case statement was previously interrupted by a break instruction.

```
switch (<expression>) {
   case <value1>:
      // statements to be executed if <expression> == <value1> : note the equality and not identity here; it uses loose comparison
      break;
   case <value2>:
   case <value3>:
      // statements to be executed if <expression> == (<value2> OR <value3>) : waterfall behavior
      break;
   default:
      // statements to be executed if <expression> didn't match any of the previous case values.
}
```

PHP Basics : for… loop Statement

PHP supports the standard for… loop statement. The for loop takes three expressions separated by semicolons: a beginning statement, a conditional and an iterative statement. At the beginning of the loop, the beginning statement is evaluated once (executed). Then the conditional is, and if it evaluates to a true value, then the statement(s) inside the for block are executed. Once the block has been executed, the iterative statement is evaluated (executed). The loop then re-evaluates the conditional statement and re-executes the for statement block if it still evaluates to true. The loop keeps doing this, including always executing the iterative statement after each iteration, until the conditional statement evaluates to false.

```
for (<start expression>; <conditional expression>; <iterative expression>) {
   // statements to execute for each iteration as long as the <conditional expression> remains true
}

for ($i = 0; $i < 5, $i++) {
   // statements block will execute 5 times sequentially for $i being equals to 0 then 1, 2, 3 and 4. at the end of the 5th iteration, $i will become 5 and
   // will cause the condition $i < 5 to become false, thus ending the loop.
}
```

It is possible to interrupt the loop by using the break statement inside the block. The loop immediately terminates upon reaching said break statement.

PHP Basics : foreach loop Statement

PHP also supports the foreach… loop that works with ***traversable*** structures such as arrays or objects that implement the Traversable interface. Traversing means iterating over the elements of a set of elements sequentially.

```
foreach (<traversable> as <element valiable>) {
   // statements to execute for each element. The variable will contain the current element's value
}
```

Because PHP arrays are associative, it is also possible to write foreach loops in a manner that allows easy access the element in the traversable as a key/value pair:

```
foreach (<traversable> as <key variable> => <value variable>) {
   // statements to execute for each element. The variables will take the corresponding key value and value value for the current element.
}
```

Just as with the for… loop, it is possible to interrupt the foreach loop by using the break statement inside the block. The loop immediately terminates upon reaching said break statement.

PHP Basics : while... and do...while loop Statements

Finally, PHP supports the while… loop. The while loop takes only a single conditional expression. It first evaluates the expression, and if it evaluates to true, then the loop's statement block is executed. At the end of the block, the conditional is re-evaluated. The loop will continue executing the statement block until the conditional expression evaluates to false. If the statements inside the block never affect the conditional expression, a while loop can be infinite and will keep executing until PHP runs out of memory.

```
while (<conditional expression>)  {
   // statements to repeat executing as long as the <conditional expression> is true.
}
```

The do… while loop is exactly similar to the while… one, except that it begins by executing the statement block and only after does it checks the conditional expression. So it executes the statement block at least once, even if the expression is false from the start.

```
do {
   // statements to repeat executing as long as the <conditional expression> is true. Will be executed at least once regardless.
} while (<conditional expression>);
```

Next week

- Functions
- PHP built-in functions
- User-defined functions