PHP SERIES · COURSE 1

PHP Fundamentals

Syntax, variables, control flow, loops, functions, arrays, strings, form handling, and includes - capped with a multi-file capstone project. 10 complete chapters, each with coding challenges and solutions.

10 Chapters

osztromok.com

CHAPTER 1: WHAT PHP IS - EMBEDDING, SYNTAX, ECHO/PRINT

COURSE 1 · CH 1

What PHP Is — Embedding, Syntax, echo/print

A server-side language that lives right inside your HTML — what that actually means, and your first real PHP statements

Every course so far in this curriculum — HTML, CSS — runs entirely in the browser. PHP is different: it runs **on the server**, before the page ever reaches a visitor's browser at all. By the time a browser receives the response, every bit of PHP has already executed — the visitor never sees PHP code itself, only the HTML it produced.

PHP Tags — Embedding PHP Inside HTML

```
<!DOCTYPE html>
<html>
<body>

  <h1>Welcome</h1>

  <?php
    echo "<p>This paragraph was generated by PHP.</p>";
  ?>

</body>
</html>
```

`<?php` opens a PHP block; `?>` closes it. Everything outside these tags is passed through to the browser completely untouched, as plain HTML — everything inside is executed by the server first.

WHAT THE BROWSER ACTUALLY RECEIVES

<h1>Welcome</h1>
This paragraph was generated by PHP.

THE .PHP FILE EXTENSION IS WHAT TELLS THE SERVER TO ACTUALLY RUN PHP AT ALL

A file saved as `.html` containing PHP tags won't execute them — the server simply serves the literal `<?php ... ?>` text as-is, visible to anyone viewing the page source. The file must be saved with a `.php` extension for the server to recognise and run the PHP inside it.

echo and print — Outputting Content

```php
<?php
  echo "Hello, world!";
  print "This works too.";

  // echo can take multiple comma-separated values
  echo "Multiple", " ", "parts", " combined.";
?>
```

`echo` and `print` both output text — `echo` is marginally faster and can take several comma-separated values at once; `print` only ever takes one. In practice, `echo` is by far the more commonly used of the two.

String Interpolation — Mixing Variables into Output

```php
<?php
  $name = "Philip";
  echo "Hello, $name!"; // double quotes interpolate variables directly
  echo 'Hello, $name!';     // single quotes do NOT — outputs literally "$name"
?>
```

DOUBLE QUOTES VS SINGLE QUOTES IS A REAL, PRACTICAL DISTINCTION IN PHP

Unlike many languages where quote style is purely stylistic, PHP treats double-quoted strings specially — variables inside them are automatically substituted with their value. Single-quoted strings are always taken completely literally. Both are valid; the choice matters for whether you actually want that substitution to happen.

Statements End with a Semicolon

```php
<?php
  echo "First statement"; // the semicolon marks the end of this statement
  echo "Second statement";
?>
```

Forgetting a semicolon is one of the most common beginner errors in PHP — the parser genuinely can't tell where one statement ends and the next begins without it, and the resulting error message often points at the *following* line rather than the actual missing semicolon.

Comments

```php
<?php
  // A single-line comment
  # Also a single-line comment – less common, but valid

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

Coding Challenges

CHALLENGE 1

Create a PHP file that outputs a complete HTML page with the title "My First Page" and a single paragraph saying "PHP is running on the server!" — using `echo` for the paragraph specifically, with the rest as plain HTML outside the PHP tags.

View solution

CHALLENGE 2

Create a variable called `$favouriteLanguage` set to "PHP", then echo a sentence using double-quote interpolation that includes the variable's value. Then echo the exact same sentence using single quotes, and observe the difference in the output.

View solution

CHALLENGE 3

Write a PHP block with three separate echo statements that together output three lines (use `"<br>"` at the end of each string to force a line break in the browser): your name, your favourite number, and a one-sentence comment about why you're learning PHP.

View solution

CHAPTER 1 QUICK REFERENCE

- **<?php ... ?>** — opens/closes a block of PHP code embedded within HTML
- **.php file extension** — required for the server to actually execute the PHP, not just serve it as text
- **echo / print** — output content; echo is more common and takes multiple comma-separated values
- **Double quotes interpolate variables** directly; single quotes never do
- **Every statement ends with a semicolon** — a very common source of beginner errors when forgotten
- **// and #** — single-line comments; **/* */** — multi-line comments
- **Next chapter:** variables, data types, and type juggling

CHAPTER 2: VARIABLES, DATA TYPES, AND TYPE JUGGLING

COURSE 1 · CH 2

Variables, Data Types, and Type Juggling

Storing values, the core PHP types, and the automatic type conversion that makes PHP genuinely different from strictly-typed languages

Variables are how a script holds onto a value to use later. PHP variables are refreshingly simple to declare — no separate type declaration step required — but understanding exactly what's happening underneath that simplicity, especially around type juggling, avoids a class of subtle bug that trips up many PHP beginners.

Declaring Variables

```php
<?php
  $name = "Philip";
  $age = 35;
  $isLearning = true;
  echo "$name is $age years old.";
?>
```

Every PHP variable name starts with a dollar sign (`$`) — that's not optional punctuation, it's a required part of the variable's name. No separate "declare this as a string/integer" step exists; PHP figures out the type automatically from the assigned value.

VARIABLE NAMES ARE CASE-SENSITIVE

`$name` and `$Name` are two genuinely different variables — a frequent source of confusing "undefined variable" warnings when a typo accidentally changes the case partway through a script.

The Core PHP Data Types

TYPE	EXAMPLE
string	Text — `"Philip"`, `'PHP'`
int	Whole numbers — `42`, `-7`
float	Decimal numbers — `3.14`, `-0.5`
bool	`true` or `false`
array	Ordered collections — covered fully in Chapter 6
null	Represents "no value at all"

Checking a Variable's Type

```php
<?php
  $value = 42;
  echo gettype($value);   // "integer"
  var_dump($value);      // shows type AND value together, very useful for debugging
?>
```

`var_dump()` is genuinely one of the most-used debugging tools in everyday PHP — it shows both the type and the exact value of whatever's passed to it, which becomes especially useful once type juggling (below) makes a value's actual type less obvious from reading the code alone.

Type Juggling — PHP's Automatic Type Conversion

PHP is "loosely typed" — it automatically converts between types as needed in many contexts, rather than requiring an explicit conversion every time. This is convenient, and also the source of some of PHP's most commonly mocked quirks.

```php
<?php
  echo "5" + 3;        // 8 — the string "5" is converted to int for the addition
  echo "5" . 3;        // "53" — the . operator concatenates, treating 3 as a string inste
  echo true + 1;        // 2 — true becomes 1, false becomes 0 in numeric contexts
  var_dump(0 == "0");    // true — loosely compared, juggled to match
```

```
  var_dump(0 === "0");   // false — strictly compared, different types, no juggling
?>
```

== VS === IS THE SINGLE MOST IMPORTANT TYPE-JUGGLING DISTINCTION TO INTERNALISE EARLY

`==` ("loose equality") allows type juggling before comparing — `0 == "abc"` historically even evaluated to `true` in older PHP versions, a famously confusing quirk. `===` ("strict equality") requires both the value *and* the type to match exactly, with zero juggling involved. As a strong default habit, prefer `===` unless you have a specific, deliberate reason to allow type juggling in a comparison.

Explicit Type Casting — When You Want Control

```
<?php
  $str = "42";
  $num = (int)$str;    // explicitly cast to integer, rather than relying on automatic jug
  $num2 = 3.7;
  $asInt = (int)$num2;  // 3 — casting float to int truncates, doesn't round
?>
```

CASTING (FLOAT)3.7 TO INT TRUNCATES, IT DOESN'T ROUND

`(int)3.7` gives `3`, not `4` — a genuinely easy assumption to get wrong if you expect mathematical rounding. Use `round()` explicitly first if rounding (rather than truncation) is actually what's needed.

Constants

```
<?php
  define('SITE_NAME', 'osztromok.com');
  echo SITE_NAME; // no $ prefix for constants, and the value can never change after defi
?>
```

A constant, once defined, cannot be reassigned later in the script — genuinely useful for values that should never change, like a site name or a fixed configuration value.

Coding Challenges

CHALLENGE 1

Create three variables: `$firstName` (string), `$age` (int), and `$heightInMeters` (float). Use `var_dump()` on each one to confirm PHP correctly identified each type, then echo a sentence combining all three using string interpolation.

View solution

CHALLENGE 2

Predict, then verify with var_dump, the result of each: `"10" + 5`, `"10" . 5`, `"3.5" + "1.5"`, and `(int)"7 apples"`. Write a one-sentence explanation for each result.

View solution

CHALLENGE 3

Write code comparing the integer `1` and the string `"1"` using both `==` and `===`, echoing a clear message explaining what each comparison returned and why they differ.

View solution

CHAPTER 2 QUICK REFERENCE

- **$variableName** — the $ prefix is required; no separate type declaration needed
- **Variable names are case-sensitive** — $name and $Name are different variables
- **Core types:** string, int, float, bool, array, null
- **gettype() / var_dump()** — check a variable's actual type; var_dump shows type and value together
- **Type juggling** — PHP automatically converts types as needed in many contexts (arithmetic, comparisons)

- **== (loose) vs === (strict)** — prefer === as a default habit to avoid juggling surprises
- **(int)/(float)/(string)** — explicit casting; casting float to int truncates, doesn't round
- **define('NAME', value)** — constants, no $ prefix, value never changes once set
- **Next chapter:** operators and control structures — if/else, switch

CHAPTER 3: OPERATORS AND CONTROL STRUCTURES

COURSE 1 · CH 3

Operators and Control Structures

Comparison and logical operators, if/elseif/else, and switch — the constructs that let a script make decisions

Chapter 2 covered comparing values with `==` and `===`. This chapter covers the full range of operators PHP offers, and the control structures that actually use them to change what a script does based on those comparisons.

Comparison Operators

OPERATOR	MEANING
== / ===	Equal (loose, juggles types) / Identical (strict, no juggling)
!= / !==	Not equal (loose) / Not identical (strict)
> / < / >= / <=	Greater than, less than, and their "or equal" variants
<=>	Spaceship operator — returns -1, 0, or 1 for less/equal/greater, genuinely useful in custom sorting

Logical Operators

```php
<?php
  $age = 25;
  $hasLicense = true;
  var_dump($age >= 18 && $hasLicense);  // true — both conditions must be true
  var_dump($age < 18 || $hasLicense);   // true — at least one condition is true
  var_dump(!$hasLicense);              // false — negates the value
?>
```

`&&` (AND), `||` (OR), and `!` (NOT) combine and negate boolean expressions — the building blocks behind any condition more complex than a single comparison.

if / elseif / else

```php
<?php
  $score = 75;
  if ($score >= 90) {
    echo "Grade: A";
  } elseif ($score >= 70) {
    echo "Grade: B";
  } elseif ($score >= 50) {
    echo "Grade: C";
  } else {
    echo "Grade: F";
  }
?>
```

PHP checks each condition in order, top to bottom, and runs the block belonging to the first one that evaluates to `true` — every subsequent `elseif` / `else` is skipped entirely once a match is found, even if a later condition would also have been true.

> **ELSEIF IS ONE WORD IN PHP — ELSE IF (TWO WORDS) ALSO WORKS, BUT MEANS SOMETHING SUBTLY DIFFERENT**
>
> `elseif` and `else if` behave identically in almost every practical case, but technically `else if` is actually an `else` block containing a nested `if` statement — relevant specifically when using PHP's alternative colon syntax, where only the single-word `elseif` is valid. Sticking with `elseif` consistently avoids needing to remember this distinction at all.

The Ternary Operator — A Compact if/else

```php
<?php
  $age = 20;
  $status = ($age >= 18) ? "adult" : "minor";
  echo $status; // "adult"

  // The null coalescing operator — a common, related shorthand
  $username = $_GET['user'] ?? 'Guest'; // use 'Guest' if $_GET['user'] isn't set
?>
```

The ternary operator condenses a simple if/else assignment into one line. The null coalescing operator (`??`) is a closely related, extremely common shorthand specifically for "use this value, or a fallback if it doesn't exist/is null" — genuinely useful and previewed here ahead of full coverage when `$_GET` / `$_POST` are properly introduced in Chapter 8.

switch — Comparing One Value Against Several Options

```php
<?php
  $day = "Wednesday";
  switch ($day) {
    case "Monday":
      echo "Start of the week";
      break;
    case "Wednesday":
      echo "Midweek";
      break;
    case "Friday":
      echo "Almost the weekend";
      break;
    default:
      echo "Just another day";
  }
?>
```

FORGETTING BREAK IS ONE OF THE MOST COMMON SWITCH MISTAKES — EXECUTION "FALLS THROUGH" TO THE NEXT CASE

Without a `break`, PHP continues executing every subsequent case's code, regardless of whether its own condition matched — this "fall-through" behaviour is occasionally used deliberately (multiple case labels sharing one block of code), but is far more often an accidental bug when a break is simply forgotten on a case that genuinely needed one.

When switch Is Better Than a Long elseif Chain

Both accomplish the same thing for comparing one variable against many possible exact values — `switch` is generally considered more readable once there are more than roughly 3-4 options, since it avoids repeating the variable name and comparison operator in every single condition.

Coding Challenges

CHALLENGE 1

Write an if/elseif/else chain that takes a temperature in Celsius (use a variable, try a few different values) and echoes "Freezing", "Cold", "Mild", or "Hot" based on reasonable thresholds you choose yourself.

View solution

CHALLENGE 2

Using the ternary operator, write a single line that assigns "Pass" or "Fail" to a variable based on whether a score variable is 50 or above. Then write the equivalent using a full if/else block, and compare the two.

View solution

CHALLENGE 3

Write a switch statement that takes a variable holding a month number (1-12) and echoes the correct season ("Winter", "Spring", "Summer", "Autumn") — group multiple case labels together for months sharing a season, using fall-through deliberately.

View solution

CHAPTER 3 QUICK REFERENCE

- **Comparison operators:** ==/===, !=/!==, >/</>=/<=, <=> (spaceship)
- **Logical operators:** && (AND), || (OR), ! (NOT)
- **if/elseif/else** — runs the FIRST matching block only, top to bottom, skipping the rest
- **Ternary (cond ? a : b)** — compact if/else for simple assignments
- **?? (null coalescing)** — value, or a fallback if it's null/unset; full coverage in Chapter 8
- **switch/case/break/default** — compares one value against several exact options
- **Forgetting break** causes fall-through — occasionally deliberate, more often an accidental bug

- **Next chapter:** loops — for, while, foreach

CHAPTER 4: LOOPS - FOR, WHILE, AND FOREACH

COURSE 1 · CH 4

Loops: for, while, and foreach

Repeating code without repeating yourself — and the specific loop each situation calls for

Loops repeat a block of code until some condition stops them. PHP offers several loop types, and most beginner confusion isn't about syntax — it's about knowing which loop fits a given situation. This chapter covers all three main types and gives a clear rule of thumb for choosing between them.

for — When You Know How Many Times to Repeat

```php
<?php
  for ($i = 1; $i <= 5; $i++) {
    echo "Count: $i<br>";
  }
?>
```

A `for` loop has three parts separated by semicolons: a starting value (`$i = 1`), a continue-while condition (`$i <= 5`), and a step to run after each iteration (`$i++`). It's the natural choice whenever the number of repetitions is known or calculable ahead of time.

while — When You Don't Know How Many Times, Just the Stop Condition

```php
<?php
  $attempts = 0;
  $found = false;
  while (!$found && $attempts < 3) {
    $attempts++;
    echo "Attempt $attempts<br>";
    // imagine $found gets set true by some real check here
```

```
  }
?>
```

`while` checks its condition before each iteration and keeps going as long as it's true — well suited to situations like "keep retrying until successful, or until a max attempt count is hit," where the exact number of iterations isn't known in advance.

do...while — The "Check After" Variant

```
<?php
  $count = 10;
  do {
    echo "This runs at least once, even though $count < 5 is false<br>";
  } while ($count < 5);
?>
```

`do...while` checks its condition *after* running the block, guaranteeing the block executes at least once regardless of the condition — genuinely useful for things like "show a menu, then keep re-showing it until the user picks 'quit'."

foreach — Purpose-Built for Arrays

```
<?php
  $colours = ["red", "green", "blue"];
  foreach ($colours as $colour) {
    echo "$colour<br>";
  }

  // With keys too — genuinely common for associative arrays
  $prices = ["apple" => 0.50, "banana" => 0.30];
  foreach ($prices as $item => $price) {
    echo "$item costs £$price<br>";
  }
?>
```

`foreach` is the idiomatic way to iterate over an array in PHP — there's no manual index counter to manage, and the `as $key => $value` form gives access to both the key and the value together. Arrays themselves are covered fully in Chapter 6; this is enough syntax to start using `foreach` productively now.

CHOOSING THE RIGHT LOOP IS MOSTLY ABOUT THE QUESTION YOU'RE ANSWERING

"Repeat exactly N times" → `for`. "Repeat until some condition becomes true/false, unknown count" → `while`. "Do this once, then keep repeating until done" → `do...while`. "Go through every item in this array" → `foreach`. Reaching for the right one by habit, rather than forcing every situation into a `for` loop, makes code noticeably more readable.

break and continue

```php
<?php
  for ($i = 1; $i <= 10; $i++) {
    if ($i == 7) {
      break;        // exits the loop entirely
    }
    if ($i % 2 == 0) {
      continue;     // skips to the next iteration, without printing
    }
    echo "$i<br>";
  }
  // prints 1, 3, 5 — stops completely once it reaches 7
?>
```

`break` exits the loop immediately, skipping any remaining iterations entirely. `continue` skips only the rest of the current iteration and moves on to the next one — the loop itself keeps running.

INFINITE LOOPS ARE A GENUINELY COMMON BEGINNER MISTAKE

A `while` loop whose condition never becomes false — often because the variable being checked is never updated inside the loop body — will run forever, typically freezing the script until PHP's execution time limit kills it. Always double-check that whatever the loop condition depends on is actually being changed somewhere inside the loop.

Comparing the Loop Types

LOOP	CHECKS CONDITION	BEST FOR
for	Before each iteration	Known/calculable repeat count
while	Before each iteration	Unknown count, condition-driven
do...while	After each iteration	Must run at least once
foreach	Per array element	Iterating over arrays

Coding Challenges

CHALLENGE 1

Use a `for` loop to echo the multiplication table for 7, from 7×1 up to 7×12, one line per result.

View solution

CHALLENGE 2

Use a `while` loop to echo every even number from 2 to 20. Then rewrite the same logic using `continue` inside a `for` loop that checks every number from 1 to 20, skipping the odd ones.

View solution

CHALLENGE 3

Create an array of 5 favourite foods. Use `foreach` to echo each one numbered (1. Pizza, 2. Sushi, etc.) — you'll need a counter variable you increment yourself, since plain foreach on a simple array doesn't give you a position number directly.

View solution

CHAPTER 4 QUICK REFERENCE

- **for (init; condition; step)** — known/calculable repeat count
- **while (condition)** — checks before each run; unknown count, condition-driven
- **do { } while (condition);** — checks after; guarantees at least one run
- **foreach ($array as $value)** / as $key => $value — idiomatic array iteration
- **break** — exits the loop entirely; **continue** — skips to the next iteration
- **Infinite loops** — happen when a while condition's variable is never updated inside the loop
- **Next chapter:** functions — defining, parameters, return values, scope

COURSE 1 · CH 5

Functions: Defining, Parameters, Return Values, and Scope

Packaging reusable logic into named blocks — and the variable-visibility rules that come with them

Every chapter so far has written code directly in the main flow of the script. Functions let you package a block of logic under a name, run it whenever needed (possibly many times, possibly with different inputs), and keep the main script focused on the bigger picture rather than repeated detail.

Defining and Calling a Function

```php
<?php
  function greet() {
    echo "Hello there!";
  }
  greet();   // calls the function, runs its body — outputs "Hello there!"
  greet();   // can be called again, as many times as needed
?>
```

A function is defined once with the `function` keyword and a name, then "called" by writing that name followed by parentheses wherever its logic is needed.

Parameters — Passing Values In

```php
<?php
  function greet($name) {
    echo "Hello, $name!<br>";
  }
  greet("Philip");   // "Hello, Philip!"
  greet("Sam");      // "Hello, Sam!"
```

```php
  // Default parameter values — used when the argument is omitted
  function greetWithDefault($name = "Guest") {
    echo "Welcome, $name!<br>";
  }
  greetWithDefault();          // "Welcome, Guest!"
  greetWithDefault("Ana");     // "Welcome, Ana!"
?>
```

Parameters let the same function behave differently depending on what's passed in. A default value (`$name = "Guest"`) makes that parameter optional — if no argument is supplied for it, the default is used instead.

Return Values — Getting a Result Back Out

```php
<?php
  function addNumbers($a, $b) {
    return $a + $b;
  }
  $total = addNumbers(4, 9);
  echo $total;   // 13
?>
```

RETURN IMMEDIATELY EXITS THE FUNCTION — ANY CODE AFTER IT NEVER RUNS

Unlike `echo`, which just prints output and continues, `return` hands a value back to wherever the function was called from *and* stops the function's execution immediately. Code placed after a `return` statement inside the same block is unreachable and will never execute.

Type Hints and Return Types — Optional, but Genuinely Useful

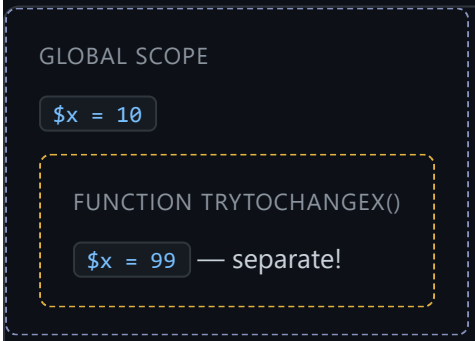
```php
<?php
  function addNumbers(int $a, int $b): int {
    return $a + $b;
  }
?>
```

Adding `int` before each parameter, and `: int` after the parentheses, tells PHP exactly what types are expected in and out. This is entirely optional in PHP (unlike strictly-typed languages where it's mandatory), but it documents intent clearly and helps PHP catch type mistakes earlier rather than letting an unexpected type silently cause confusing behaviour deeper in the script.

Variable Scope — Where a Variable "Exists"

```php
<?php
  $x = 10;   // a global-scope variable
  function tryToChangeX() {
    $x = 99;   // this is a DIFFERENT $x, local to this function only
    echo $x;       // 99
  }
  tryToChangeX();
  echo $x;          // still 10 — the outer $x was never touched
?>
```

Every function has its own local scope — variables created inside a function exist only inside that function, completely separate from a same-named variable outside it. This is a deliberate safety feature, not a limitation: it means a function's internal variables can never accidentally clash with or overwrite variables elsewhere in the script.



Two completely separate `$x` variables — the function's local `$x` has no effect on the global one.

GLOBAL KEYWORD AND FUNCTION ARGUMENTS ARE THE TWO REAL WAYS DATA CROSSES THE SCOPE BOUNDARY

A function can access an outer variable deliberately by declaring `global $x;` as its first line — but this is rarely the cleanest approach in practice. Passing the value in as a parameter, and getting a

result back via `return`, is almost always the better, more predictable pattern, and is the approach used throughout this course.

Coding Challenges

CHALLENGE 1

Write a function `isEven($number)` that returns `true` if a number is even and `false` if it's odd (using the `%` remainder operator from Chapter 4). Call it with a few different numbers and echo the results using var_dump.

View solution

CHALLENGE 2

Write a function `calculateRectangleArea($width, $height = 1)` with a default height of 1, so it can also be used to calculate the area of a square just by passing one argument. Call it three different ways and echo each result.

View solution

CHALLENGE 3

Create a global variable `$counter = 0`. Write a function that creates its own local variable also called `$counter`, sets it to 100, and echoes it. After calling the function, echo the original global `$counter` to demonstrate it was never affected — add a comment explaining why.

View solution

CHAPTER 5 QUICK REFERENCE

- **function name(params) { }** — defines a reusable named block of code
- **Default parameters** — function f($x = "default") makes an argument optional
- **return** — hands a value back AND exits the function immediately; code after it never runs
- **echo vs return** — echo prints output, return hands back a usable value to the caller

- **Type hints** — optional but recommended: function f(int $x): int
- **Scope** — variables inside a function are local; they cannot accidentally affect same-named outer variables
- **global keyword** — exists, but passing parameters and using return is the cleaner default pattern
- **Next chapter:** arrays — indexed, associative, and common array functions

CHAPTER 6: ARRAYS - INDEXED, ASSOCIATIVE, AND FUNCTIONS

COURSE 1 · CH 6

Arrays: Indexed, Associative, and Common Array Functions

Storing collections of values — by position or by name — and the built-in functions for working with them

Arrays appeared briefly in Chapter 4's `foreach` examples. This chapter covers them properly: the two main styles PHP arrays come in, and the handful of built-in functions used constantly in real PHP code.

Indexed Arrays — Position-Based

```php
<?php
  $fruits = ["apple", "banana", "cherry"];
  echo $fruits[0];   // "apple" — indexing starts at 0, not 1
  echo $fruits[2];   // "cherry"
  $fruits[] = "date";       // appends to the end — now index 3
  print_r($fruits);          // dumps the whole array readably
?>
```

PHP ARRAYS ARE ZERO-INDEXED — THE FIRST ELEMENT IS INDEX 0, NOT 1

This trips up plenty of beginners coming from everyday counting (where you'd naturally call "apple" the "first" item and expect it at position 1). `$fruits[0]` is the first element; `$fruits[count($fruits) - 1]` is the last.

Associative Arrays — Named Keys

```php
<?php
  $person = [
    "name" => "Philip",
    "age" => 35,
    "city" => "London"
```

```
  ];
  echo $person["name"];   // "Philip"
  $person["email"] = "philip@example.com";  // adds a new key
?>
```

An associative array uses meaningful string keys instead of numeric positions — genuinely more readable than remembering "index 1 is always the age" when modelling real-world data like a person's details.

Multidimensional Arrays — Arrays of Arrays

```
<?php
  $people = [
    ["name" => "Philip", "age" => 35],
    ["name" => "Sam",    "age" => 28]
  ];
  echo $people[0]["name"];   // "Philip"
  foreach ($people as $person) {
    echo $person["name"] . " is " . $person["age"] . "<br>";
  }
?>
```

An array's elements can themselves be arrays — extremely common for representing lists of records, like rows that might later come from a database (covered in a future course on database integration).

Common Built-In Array Functions

FUNCTION	WHAT IT DOES
count($arr)	Number of elements
array_push($arr, $val)	Adds to the end (same as $arr[] = $val)
array_pop($arr)	Removes and returns the last element
array_merge($a, $b)	Combines two arrays into one

FUNCTION	WHAT IT DOES
in_array($val, $arr)	Checks whether a value exists in the array
array_search($val, $arr)	Returns the key/index of a value, or false if not found
sort($arr)	Sorts values, re-indexes keys from 0
array_map($fn, $arr)	Applies a function to every element, returns a new array
array_filter($arr, $fn)	Keeps only elements where the function returns true

```php
<?php
  $numbers = [1, 2, 3, 4, 5];

  // array_map — transform every element
  $doubled = array_map(function($n) { return $n * 2; }, $numbers);
  print_r($doubled);   // [2, 4, 6, 8, 10]

  // array_filter — keep only matching elements
  $evens = array_filter($numbers, function($n) { return $n % 2 == 0; });
  print_r($evens);     // [1 => 2, 3 => 4] — note original keys are preserved!
?>
```

ARRAY_MAP AND ARRAY_FILTER TAKE AN ANONYMOUS FUNCTION AS AN ARGUMENT

The `function($n) { ... }` passed in is a function with no name, defined right where it's used — this pattern (a "callback") is genuinely common in PHP once you start using these array functions. It will feel more natural with practice; for now, treat it as "the rule to apply to each element," written inline rather than as a separately-named function.

ARRAY_FILTER PRESERVES ORIGINAL KEYS — THIS SURPRISES PEOPLE EXPECTING A CLEAN 0,1,2... RESULT

After filtering, the remaining elements keep whatever index/key they originally had, leaving gaps. If a clean re-indexed array is needed afterward, wrap the result in `array_values()`.

Coding Challenges

CHALLENGE 1

Create an associative array representing a book with keys "title", "author", and "year". Echo a formatted sentence using all three values, then add a new key "genre" and echo the whole array with print_r.

View solution

CHALLENGE 2

Given an indexed array of 6 numbers of your choice, use in_array() to check if 42 is present, array_search() to find the index of a number you know is in there, and sort() to put the array in ascending order. Echo the result of each step.

View solution

CHALLENGE 3

Given an array of 8 numbers, use array_filter() with an anonymous function to keep only numbers greater than 10, then use array_map() on the filtered result to square each remaining number. Wrap the filtered result in array_values() before mapping, and print_r the final array.

View solution

CHAPTER 6 QUICK REFERENCE

- **Indexed arrays** — zero-based numeric positions: $arr[0] is the first element
- **Associative arrays** — named string keys: $arr["key"]
- **Multidimensional arrays** — arrays of arrays, accessed like $arr[0]["key"]
- **count(), array_push/pop, array_merge, in_array, array_search, sort** — everyday array operations
- **array_map($fn, $arr)** — transforms every element, returns a new array
- **array_filter($arr, $fn)** — keeps only matching elements, preserves original keys
- **array_values($arr)** — re-indexes an array from 0, useful after filtering
- **Next chapter:** strings — common string functions, formatting, and manipulation

COURSE 1 · CH 7

Strings: Functions, Formatting, and Manipulation

The built-in toolkit for measuring, searching, slicing, and reshaping text

Strings have appeared in every chapter so far, but always as fixed pieces of text. This chapter covers PHP's large built-in library of string functions — the everyday tools for working with text that isn't already in exactly the shape you need.

Length and Case

```php
<?php
  $text = "Hello, PHP!";
  echo strlen($text);          // 11 — number of characters
  echo strtoupper($text);      // "HELLO, PHP!"
  echo strtolower($text);      // "hello, php!"
  echo ucfirst("hello");        // "Hello" — capitalises only the first letter
  echo ucwords("hello there");  // "Hello There" — capitalises every word
?>
```

Searching Inside a String

```php
<?php
  $sentence = "The quick brown fox";
  var_dump(str_contains($sentence, "brown"));  // true
  echo strpos($sentence, "brown");          // 10 — the starting position, or false if not
  var_dump(str_starts_with($sentence, "The")); // true
?>
```

STRPOS CAN RETURN 0 — AND 0 IS "FALSY", SO USE === FOR THE NOT-FOUND CHECK

If the search text is found right at the very start of the string, `strpos()` correctly returns `0`. But `0` behaves as "false-ish" in a loose boolean check, making `if (!strpos(...))` wrongly treat "found at position 0" the same as "not found at all". Always write `strpos($str, $search) === false` to test specifically for "not found", never a plain truthy/falsy check.

Slicing and Replacing

```php
<?php
  $text = "Hello, World!";
  echo substr($text, 7);        // "World!" — from position 7 to the end
  echo substr($text, 7, 5);     // "World" — 5 characters starting at position 7
  echo str_replace("World", "PHP", $text); // "Hello, PHP!"
  echo trim("  padded text  ");    // "padded text" — removes leading/trailing whitespace
?>
```

Splitting and Joining

```php
<?php
  $csv = "apple,banana,cherry";
  $parts = explode(",", $csv);   // ["apple", "banana", "cherry"]
  print_r($parts);

  $joined = implode(" - ", $parts);  // "apple - banana - cherry"
  echo $joined;
?>
```

`explode()` turns a string into an array by splitting on a delimiter; `implode()` does the reverse — joining an array's elements into a single string with a chosen separator between them.

Formatting Output — sprintf

```php
<?php
  $price = 9.5;
  $formatted = sprintf("Price: £%.2f", $price);
  echo $formatted;   // "Price: £9.50" — forces exactly 2 decimal places
```

```php
    $name = "Sam";
    $age = 7;
    echo sprintf("%s is %d years old", $name, $age);  // "Sam is 7 years old"
?>
```

`sprintf()` builds a formatted string using placeholders — `%s` for strings, `%d` for integers, `%.2f` for a float fixed to 2 decimal places. It's especially useful for money values, where plain string interpolation can't reliably guarantee a consistent number of decimal places.

FUNCTION	WHAT IT DOES
strlen($s)	Length in characters
strtoupper / strtolower	Change case
str_contains / strpos	Search inside a string
substr($s, $start, $len)	Extract part of a string
str_replace($search, $replace, $s)	Replace occurrences
trim($s)	Remove leading/trailing whitespace
explode($delim, $s)	String → array
implode($glue, $arr)	Array → string
sprintf($format, ...)	Build a precisely formatted string

SINGLE QUOTES VS DOUBLE QUOTES ALSO MATTERS FOR STRLEN() WITH SPECIAL CHARACTERS — BUT FOR EVERYDAY TEXT, THE FUNCTION LIBRARY ABOVE WORKS THE SAME REGARDLESS OF QUOTE STYLE

Most of the time it doesn't matter whether the original string used single or double quotes — once it's stored in a variable, functions like `strlen()` or `strtoupper()` treat it identically either way.

Coding Challenges

CHALLENGE 1

Take the string " philip osztromok " (note the extra spaces). Trim it, then use ucwords() to capitalise each word properly, and echo the final cleaned-up result along with its length using strlen().

View solution

CHALLENGE 2

Given the string "2026-06-20", use explode() to split it into year, month, and day parts. Then use sprintf() to echo it back out in the format "20/06/2026" (day/month/year, UK style).

View solution

CHALLENGE 3

Write a function censorWord($sentence, $word) that uses str_replace() to replace every occurrence of $word in $sentence with asterisks matching its length (e.g. "cat" becomes "***"). Test it on a sentence containing the target word twice.

View solution

CHAPTER 7 QUICK REFERENCE

- **strlen, strtoupper/strtolower, ucfirst, ucwords** — length and case
- **str_contains, strpos, str_starts_with** — searching; use === false to test "not found" with strpos
- **substr($s, $start, $len)** — extract a portion of a string
- **str_replace, trim** — replacing and cleaning up whitespace
- **explode / implode** — string ↔ array conversions
- **sprintf($format, ...)** — precise formatted output, especially for numbers/money
- **Next chapter:** superglobals — $_GET, $_POST, and $_SERVER, and handling form input

CHAPTER 8: SUPERGLOBALS - $_GET, $_POST, $_SERVER

COURSE 1 · CH 8

Superglobals: $_GET, $_POST, and $_SERVER

Reading data that comes from outside your script — the URL, a submitted form, and the request itself

Every PHP script so far has run in isolation, with all its data defined inside the file itself. Real web pages need to receive information from the outside world — a search box's typed text, a login form's fields, which page a link was clicked from. PHP hands you this information through a set of special, automatically-populated arrays called **superglobals** — available in every scope, with no `global` keyword needed to reach them.

$_GET — Data in the URL

When a URL contains a query string like `page.php?name=Sam&age=7`, PHP automatically parses everything after the `?` into the `$_GET` array.

```
// URL: page.php?name=Sam&age=7

<?php
  echo $_GET['name'];  // "Sam"
  echo $_GET['age'];   // "7" — note: always a string, even though it looks numeric
?>
```

EVERYTHING FROM $_GET AND $_POST ARRIVES AS A STRING

Even `age=7` in the URL comes through as the string `"7"`, not the integer `7`. If you're going to do arithmetic with it, cast it explicitly first — `(int) $_GET['age']` — the same type-juggling behaviour from Chapter 2 applies here too, since PHP will still happily compare `"7" == 7` as true, but won't automatically treat it as a true int everywhere.

$_POST — Data From a Submitted Form

A form using `method="post"` sends its field values in the request body rather than the URL — appropriate for anything longer, more sensitive, or that shouldn't sit visibly in a browser history or bookmark. PHP parses these into `$_POST` the same way it parses `$_GET`.

```
<!-- A simple HTML form -->
<form method="post" action="process.php">
  <input type="text" name="username">
  <input type="submit">
</form>
```

```
// process.php
<?php
  $name = $_POST['username'];
  echo "Hello, " . $name . "!";
?>
```

Handling a Missing Key Safely

Accessing `$_GET['name']` when no `name` parameter was actually sent triggers an "undefined array key" warning and evaluates to `null`. Since user input is never guaranteed to include every field you expect, always check first.

```
<?php
  // Safe pattern using isset()
  $name = isset($_GET['name']) ? $_GET['name'] : 'Guest';

  // Cleaner equivalent using the null coalescing operator
  $name = $_GET['name'] ?? 'Guest';

  echo "Hello, " . $name;
?>
```

The `??` null coalescing operator returns its left-hand value if it exists and isn't `null`, otherwise it falls back to the right-hand default — a compact replacement for the longer `isset() ? ... : ...` pattern, and the standard, idiomatic way to safely read superglobal data in modern PHP.

$_SERVER — Information About the Request Itself

`$_SERVER` is populated automatically by PHP with details about the server environment and the current request — no form or URL parameter needed to fill it.

```php
<?php
  echo $_SERVER['REQUEST_METHOD'];  // "GET" or "POST"
  echo $_SERVER['PHP_SELF'];       // the currently executing script's own path
  echo $_SERVER['HTTP_USER_AGENT']; // the visitor's browser identification string
?>
```

A very common pattern is branching behaviour based on `REQUEST_METHOD` — showing a blank form on a first, plain `GET` visit to a page, then processing the submitted data only once the same page receives a `POST`:

```php
<?php
  if ($_SERVER['REQUEST_METHOD'] === 'POST') {
      $name = $_POST['username'] ?? '';
      echo "Thanks, " . $name;
  } else {
      echo '<form method="post"><input name="username"><input type="submit"></form>';
  }
?>
```

$_GET, $_POST, AND $_SERVER ARE STILL ORDINARY ARRAYS

Every array function from Chapter 6 works on them directly — `count($_GET)` tells you how many query parameters arrived, `array_keys($_POST)` lists every submitted field name, and a `foreach` loop can walk through every value in either without knowing the field names in advance.

SUPERGLOBAL	POPULATED FROM
$_GET	Query string parameters in the URL (?key=value)
$_POST	Submitted form fields, when method="post"
$_SERVER	Request/environment info (method, script path, headers)
$_SERVER['REQUEST_METHOD']	"GET" or "POST" for the current request

Coding Challenges

CHALLENGE 1

Simulate a $_GET array manually (since you're not running this through a real web server yet) with keys "product" and "qty". Safely read both using the ?? operator with sensible defaults, and echo a formatted sentence using the values.

View solution

CHALLENGE 2

Write a small script that checks $_SERVER['REQUEST_METHOD']. If it equals "POST", echo a thank-you message using a simulated $_POST['email'] value; otherwise, echo a message telling the visitor to submit the form first.

View solution

CHALLENGE 3

Given a simulated $_POST array representing a contact form (name, email, message — but with "email" deliberately missing), loop over $_POST with foreach and print_r any keys that are empty or missing, using isset() to check each expected field safely.

View solution

CHAPTER 8 QUICK REFERENCE

- **$_GET** — data from a URL's query string; always strings
- **$_POST** — data from a submitted form using method="post"
- **$_SERVER** — info about the request itself (method, script path, headers)
- **?? (null coalescing)** — the safe, idiomatic way to read a superglobal key that might not exist
- **$_SERVER['REQUEST_METHOD']** — the standard way to branch between "show the form" and "process the form" on the same page

- **Next chapter:** splitting code across multiple files with include and require

CHAPTER 9: INCLUDES - INCLUDE, REQUIRE, ORGANISING CODE

COURSE 1 · CH 9

Splitting Code Across Files: include and require

Reusing shared PHP code — headers, footers, and helper functions — across many pages without copy-pasting

Every script so far has lived in a single file. Real projects share the same header, footer, or set of helper functions across dozens of pages — copy-pasting that code into every file would mean editing every single one whenever something needs to change. PHP solves this with four statements that pull another file's own code directly into the current script: `include`, `require`, `include_once`, and `require_once`.

include and require

```php
// header.php
<?php
  echo "<header>My Website</header>";
?>
```

```php
// index.php
<?php
  include 'header.php';
  echo "<main>Welcome!</main>";
?>
```

When `index.php` runs, `include 'header.php'` pulls in and executes `header.php`'s own code right at that point in the script, exactly as if its contents had been pasted there directly. Any variables already defined before the `include` line are visible to the included file, and any variables the included file defines become visible afterward too — they share the same scope.

INCLUDE VS REQUIRE — THE DIFFERENCE IS WHAT HAPPENS WHEN THE FILE IS MISSING

If the target file doesn't exist, `include` emits a warning and the script **keeps running** — the rest of the page still renders, just missing that piece. `require` emits a fatal error and **stops the script**

immediately. Use `require` for anything the page genuinely cannot function without (a database config, a core class definition); use `include` for something optional, like a sidebar widget that's fine to simply be absent.

include_once and require_once

If two different included files both try to include a third shared file — say, a function library both of them rely on — a plain `include` would load and execute that shared file's code twice, which for a function definition causes a fatal "cannot redeclare function" error.

```php
// functions.php
<?php
  function greet($name) {
      return "Hello, " . $name . "!";
  }
?>
```

```php
// index.php
<?php
  require_once 'functions.php';
  require_once 'functions.php';  // safely skipped — already loaded

  echo greet('Sam');
?>
```

`include_once` and `require_once` track which files have already been pulled in during the current script run, and silently skip any file that's already been loaded — combining "load this file" with "but never load it twice." For a function or class library that might legitimately be included from several different places, `require_once` is the standard, safe default.

A PRACTICAL SHARED-HEADER PATTERN

A common real-world layout splits a page into `header.php` (opening HTML, navigation), the page's own unique content, and `footer.php` (closing HTML, shared scripts) — with every page on the site including the same two shared files at the top and bottom. Updating the navigation menu then means editing `header.php` once, not every page individually.

STATEMENT	ON A MISSING FILE	LOADS THE FILE AGAIN IF ALREADY INCLUDED?
include	Warning, script continues	Yes
require	Fatal error, script stops	Yes
include_once	Warning, script continues	No — safely skipped
require_once	Fatal error, script stops	No — safely skipped

Coding Challenges

CHALLENGE 1

Describe (in comments, since this is a single-file exercise) a two-file setup: greeting.php defines a function sayHello($name), and main.php uses require_once to load it and calls sayHello() with your own name. Write out both files' full code as comments to show the structure.

View solution

CHALLENGE 2

Explain, in your own words as a comment, what would happen if index.php used require 'config.php' but config.php didn't exist, versus what would happen if it used include 'config.php' instead. Write a short code example demonstrating the require case.

View solution

CHALLENGE 3

Write out (as comments describing a 3-file structure) why using require_once instead of require would prevent a "cannot redeclare function" fatal error if both header.php and sidebar.php each try to require the same functions.php file from within index.php.

View solution

CHAPTER 9 QUICK REFERENCE

- **include** — pulls in a file; warning + continues if missing
- **require** — pulls in a file; fatal error + stops if missing
- **include_once / require_once** — same as above, but skips a file already loaded once
- An included file shares scope with the file that included it — variables flow both ways
- Use require_once for anything the page can't function without, especially shared function/class libraries
- Use include for genuinely optional pieces, like an optional sidebar widget
- **Next:** the Capstone — combining variables, functions, arrays, strings, and includes into one multi-file project

CAPSTONE: A MULTI-FILE PROFILE CARD GENERATOR

COURSE 1 · CAPSTONE

Capstone: A Multi-File Profile Card Generator

Bringing together variables, control structures, functions, arrays, strings, superglobals, and includes into one small real project

Every chapter in this course introduced one piece in isolation. This capstone project pulls all of them together into a single small, working program: a **profile card generator** that takes a person's details from a simulated form submission and produces a formatted, styled summary — split across multiple files, the way a real project would be organised.

The Project Structure

The project is split into three files, each with a single clear job — mirroring the shared-header pattern from Chapter 9:

- `helpers.php` — a small library of reusable functions (formatting, validation)
- `profile_card.php` — the function that builds a formatted profile card from a data array
- `index.php` — reads the "submitted" data, includes the other two files, and renders the final output

helpers.php

```php
<?php
  // A small reusable helper library — Chapter 5's function/scope material,
  // Chapter 7's string functions

  function formatName($name) {
      return ucwords(trim($name));
  }

  function isValidField($value) {
      return isset($value) && $value !== '';
  }
?>
```

profile_card.php

```php
<?php
  // Builds one formatted profile card — Chapter 6's array handling,
  // Chapter 3's control structures, Chapter 7's sprintf()

  function buildProfileCard($profile) {
      $name   = formatName($profile['name'] ?? '');
      $role   = $profile['role'] ?? 'Member';
      $skills = $profile['skills'] ?? [];

      if (!isValidField($name)) {
          return "⚠ Cannot build a card — missing name.";
      }

      $skillList = count($skills) > 0
          ? implode(', ', $skills)
          : 'No skills listed';

      return sprintf(
          "%s\n%s\nSkills: %s",
          $name,
          $role,
          $skillList
      );
  }
?>
```

index.php

```php
<?php
  // The entry point — Chapter 8's superglobals, Chapter 9's require_once

  require_once 'helpers.php';
  require_once 'profile_card.php';

  // Simulating a submitted $_POST array, as if from a real form
  $_POST = [
      'name'   => '  philip osztromok  ',
      'role'   => 'Web Developer',
      'skills' => ['PHP', 'MySQL', 'JavaScript']
  ];

  $profile = [
      'name'   => $_POST['name'] ?? '',
```

```
        'role'   => $_POST['role'] ?? '',
        'skills' => $_POST['skills'] ?? []
    ];

    echo buildProfileCard($profile);
?>
```

OUTPUT

Philip Osztromok
Web Developer
Skills: PHP, MySQL, JavaScript

What Each Chapter Contributed

Nothing in the project above is new — every line traces back to a specific earlier chapter:

- **Chapter 2 (Variables & Type Juggling):** `$profile`, `$name`, `$skillList` — variables holding and passing data between functions
- **Chapter 3 (Operators & Control Structures):** the `if` check for a missing name, and the ternary `? :` deciding the skill list text
- **Chapter 5 (Functions & Scope):** `formatName()`, `isValidField()`, and `buildProfileCard()` — each with its own local scope, parameters, and return value
- **Chapter 6 (Arrays):** `$profile` and `$_POST['skills']` as associative and indexed arrays, plus `count()` and `implode()`
- **Chapter 7 (Strings):** `trim()`, `ucwords()`, `implode()`, and `sprintf()` for building the final formatted output
- **Chapter 8 (Superglobals):** reading `$_POST` safely with the `??` null coalescing operator
- **Chapter 9 (Includes):** `require_once` to pull the helper library and the card-builder function into `index.php`

WHY REQUIRE_ONCE WAS THE RIGHT CHOICE HERE, NOT INCLUDE

`index.php` genuinely cannot build a profile card without `buildProfileCard()` being defined — if `profile_card.php` failed to load, the whole page should stop with a clear error rather than silently continuing and producing a confusing "call to undefined function" failure further down. That's exactly

the `require`-over-`include` reasoning from Chapter 9, applied to a real project rather than an abstract rule.

Final Coding Challenge

FINAL CHALLENGE

Extend buildProfileCard() to also accept an optional "email" key. If present and valid (contains an @ character), append a line "Contact: {email}" to the output; if missing, append "Contact: not provided" instead. Test it with two different simulated $_POST arrays — one with a valid email, one without any email key at all — and echo both results.

View solution

Course Complete

This capstone closes out PHP Fundamentals. From here, **PHP Intermediate** builds directly on this foundation — sessions and cookies for remembering a visitor between page loads, object-oriented PHP (classes, the same concept the arrays and functions here were quietly building toward), working with a real MySQL database instead of simulated arrays, and proper input validation and sanitisation for handling real, untrusted user input safely.

COURSE 1 COMPLETE — WHAT YOU'VE LEARNED

- **Ch 1–2:** PHP syntax, echo, variables, and PHP's own type-juggling rules
- **Ch 3–4:** Operators, if/elseif/else, and all three loop types (for, while, foreach)
- **Ch 5:** Functions — parameters, return values, and global vs. local scope
- **Ch 6:** Indexed and associative arrays, plus the core array function library
- **Ch 7:** The string function library — searching, slicing, replacing, and sprintf formatting
- **Ch 8:** Reading external data safely via $_GET, $_POST, and $_SERVER
- **Ch 9:** Splitting code across files with include/require and their _once variants

- **Capstone:** a real multi-file project combining every one of the above