FREE COMPANION DOWNLOAD

TYPE · PREDICT · RUN · MODIFY

50 Extra Practice Questions

Three or four per chapter, across all fourteen chapters

From **PHP the TPRM Way** by John I. Jerkovic

PRACTICAL TECH MADE SIMPLE

WORK THROUGH THESE FIRST

Questions

1 CH 1 · PREDICT

Write down the exact three output lines. Say which statement performs arithmetic and which one does not, and why.

```php
<?php

declare(strict_types=1);

$a = 4;
$b = 3;

echo "Total:", PHP_EOL;
echo $a + $b, PHP_EOL;
echo "$a + $b", PHP_EOL;
```

2 CH 1 · DEBUG

This program will not start. Name the kind of problem, say which line PHP reports and which line is actually at fault, then make the smallest correction.

```php
<?php

declare(strict_types=1);

$title = "Profile Card"
echo $title . PHP_EOL;
```

3 CH 1 · EXPLAIN

This program is meant to print the perimeter of a 7 by 3 rectangle, which is 20. It prints 17 instead, and reports nothing. Which of the three kinds of problem is this, and why does PHP say nothing about it?

```php
<?php

declare(strict_types=1);

$width = 7;
$height = 3;
$perimeter = 2 * $width + $height;

echo "Perimeter: " . $perimeter . PHP_EOL;
```

4 CH 2 · TRACE

Write the exact three lines, including the type and length information `var_dump()` reports.

```php
<?php

declare(strict_types=1);

$countText = "12";
$countNumber = 12;

var_dump($countText === $countNumber);
var_dump((int) $countText === $countNumber);
var_dump($countText . $countNumber);
```

5 CH 2 · PREDICT

Write the four output lines exactly. Explain what the single quotation marks changed, and what the backslash on the last line is for.

```php
<?php

declare(strict_types=1);

$name = "Maya";
$item = ["title" => "Notebook"];

echo "Hello, $name." . PHP_EOL;
echo 'Hello, $name.' . PHP_EOL;
echo "Item: {$item['title']}" . PHP_EOL;
echo "Cost: \$4.50" . PHP_EOL;
```

6 CH 2 · DEBUG

The program should print the age next year. Explain what it prints instead and why, then repair it so that a value which is not a number is refused.

```php
<?php

declare(strict_types=1);

$ageText = "16";
$nextYear = $ageText . 1;

echo "Next year: " . $nextYear . PHP_EOL;
```

7 CH 3 · TRACE

Trace `$total` through the four assignments, then write all four output lines.

```php
<?php

declare(strict_types=1);

$total = 10;
$total += 5;
$total *= 2;
$total -= 4;

echo $total . PHP_EOL;
echo intdiv($total, 4) . PHP_EOL;
echo ($total % 4) . PHP_EOL;
echo ($total / 4) . PHP_EOL;
```

8 CH 3 · PREDICT

Predict all four lines. Then say why the first two disagree about the same pair of numbers.

```php
<?php

declare(strict_types=1);

$price = 0.1 + 0.2;

var_dump($price === 0.3);
var_dump(abs($price - 0.3) < 0.0001);
echo $price . PHP_EOL;
echo number_format($price, 2) . PHP_EOL;
```

9 CH 3 · BUILD

Start from an empty file. Store a number of seconds in a variable, then print it as hours, minutes, and seconds. Use integer division and the remainder operator, and give every intermediate value a name. Check your program against 3725 seconds, then against 60, then against 0.

10 CH 4 · TRACE

Write the four output lines. Three of them are wrong for a normal grade scale. Say which condition is at fault and why.

```php
<?php

declare(strict_types=1);

$scores = [95, 84, 70, 59];

foreach ($scores as $score) {
    if ($score >= 70) {
        $grade = "C";
    } elseif ($score >= 80) {
        $grade = "B";
    } elseif ($score >= 90) {
        $grade = "A";
    } else {
        $grade = "Needs improvement";
    }

    echo $score . ": " . $grade . PHP_EOL;
}
```

11 CH 4 · PREDICT

Predict all five booleans. For each one, say whether the comparison converted anything before comparing.

```php
<?php

declare(strict_types=1);

$input = "0";

var_dump($input == 0);
var_dump($input === 0);
var_dump($input === "0");
var_dump($input == false);
var_dump($input !== "");
```

12 CH 4 · DEBUG

The documented rule is that a score of 60 or more passes. Run this program in your head first, then say what it prints and what the smallest correction is.

```php
<?php

declare(strict_types=1);

$passMark = 60;
$score = 60;

if ($score > $passMark) {
    echo "Pass" . PHP_EOL;
} else {
    echo "Fail" . PHP_EOL;
}
```

13 CH 4 · MODIFY

Rewrite this chain as a match expression that produces the label rather than echoing it. Then predict what your version does with a zone of `"antarctica"`, first with a default arm and then without one.

```php
<?php

declare(strict_types=1);

$zone = "uk";

if ($zone === "domestic") {
    echo "Standard 3-5 days" . PHP_EOL;
} elseif ($zone === "eu" || $zone === "uk") {
    echo "International 5-10 days" . PHP_EOL;
} else {
    echo "Zone not served" . PHP_EOL;
}
```

14 CH 5 · TRACE

Trace `$counted` and `$total` through every pass of the loop, then write the two output lines.

```php
<?php

declare(strict_types=1);

$values = ["8", "abc", "12", "", "5"];
$total = 0;
$counted = 0;

foreach ($values as $value) {
    if (!is_numeric($value)) {
        continue;
    }

    $total += (int) $value;
    $counted += 1;
}

echo $counted . PHP_EOL;
echo $total . PHP_EOL;
```

15 CH 5 · PREDICT

Write every line this program prints, in order.

```php
<?php

declare(strict_types=1);

$found = -1;

for ($i = 0; $i < 5; $i += 1) {
    echo "checking " . $i . PHP_EOL;

    if ($i === 2) {
        $found = $i;
        break;
    }
}

echo "found: " . $found . PHP_EOL;
```

16 CH 5 · DEBUG

This loop should print all three items with numbers beside them. Say exactly what it prints, name both defects, and repair them.

```php
<?php

declare(strict_types=1);

$items = ["pen", "book", "lamp"];

for ($i = 1; $i <= count($items); $i += 1) {
    echo $i . ": " . $items[$i] . PHP_EOL;
}
```

17 CH 5 · BUILD

Start from an empty file. Put a handful of scores in an array, then report the count, the total, the lowest, the highest, and the average, using one loop and without calling `array_sum()`, `min()`, or `max()`. Handle an empty array without dividing by zero. Test it with several scores, then with one score, then with none.

18 CH 6 · TRACE

Write the three output lines, and say which parameter each call supplied and which it left to the default.

```php
<?php

declare(strict_types=1);

function invoiceLine(string $item, int $quantity = 1, float $unitPrice = 0.0): string
{
    return $item . " x" . $quantity . " = " . number_format($quantity * $unitPrice, 2);
}

echo invoiceLine("Pen") . PHP_EOL;
echo invoiceLine("Pen", 4, 1.25) . PHP_EOL;
echo invoiceLine("Pen", unitPrice: 1.25) . PHP_EOL;
```

19 CH 6 · DEBUG

A number appears on screen and `$total` is `null`. Explain both facts, then rewrite `addTax()` as a calculation function and let the caller do the printing.

```php
<?php

declare(strict_types=1);

function addTax(float $subtotal): void
{
    echo $subtotal * 1.13;
}

$total = addTax(100.0);

echo PHP_EOL;
var_dump($total);
```

20 CH 6 · EXPLAIN

This file leaves out the declare line on purpose. Predict what it prints, then add `declare(strict_types=1);` as the first statement and predict again. Explain which of the two behaviors you would rather have in a program that reads a form.

```php
<?php

// This file has no declare(strict_types=1) line, on purpose.

function isPassing(int $score): bool
{
    return $score >= 60;
}

var_dump(isPassing("72"));
```

21 CH 6 · BUILD

Start from an empty file. Write isPassing(int `$score`, int `$passMark` = 60): bool, then call it with the values 59, 60, and 61 and print each result. Say in a comment which of the three results proves the rule you documented.

22 CH 7 · TRACE

Write the three output lines. Say why the first two counts differ, and what `trim()` did and did not remove.

```php
<?php

declare(strict_types=1);

$raw = "  red , green ,, blue  ";
$parts = explode(",", trim($raw));

echo count($parts) . PHP_EOL;

$clean = [];

foreach ($parts as $part) {
    $part = trim($part);

    if ($part !== "") {
        $clean[] = $part;
    }
}

echo count($clean) . PHP_EOL;
echo implode(" | ", $clean) . PHP_EOL;
```

23 CH 7 · DEBUG

This line puts a display name into a page. Say exactly what it sends, explain what a visitor could do with that, and repair it. Then say what the repair does not do.

```php
<?php

declare(strict_types=1);

$displayName = '<b>Maya</b> & Co.';

echo "<p>Hello " . $displayName . "</p>" . PHP_EOL;
```

24 CH 7 · BUILD

Start from an empty file. Count how many times a chosen letter occurs in a sentence, without using a regular expression, and report both a case-sensitive and a case-insensitive count. State your normalization rule in a comment.

25 CH 8 · TRACE

Write the three output lines. Pay particular attention to the second one.

```php
<?php

declare(strict_types=1);

$temperatures = [18.5, 22.0, 15.5, 30.0];
$sorted = $temperatures;

sort($sorted);

echo implode(", ", $sorted) . PHP_EOL;
echo implode(", ", $temperatures) . PHP_EOL;
echo $sorted[0] . " " . $sorted[count($sorted) - 1] . PHP_EOL;
```

26 CH 8 · PREDICT

Predict all four output lines. Say which of the three array functions changed `$names` and which left it alone.

```php
<?php

declare(strict_types=1);

$names = ["Maya", "Noah", "Ava", "Sam"];

$upper = array_map("strtoupper", $names);
$short = array_filter($names, fn (string $name): bool => strlen($name) < 4);

echo implode(",", $upper) . PHP_EOL;
echo implode(",", $short) . PHP_EOL;
echo count($names) . PHP_EOL;

$last = array_pop($names);

echo $last . " " . count($names) . PHP_EOL;
```

27 CH 8 · DEBUG

The item is in the list, and the program says it is not. Explain the cause in one sentence, then make the correction.

```php
<?php

declare(strict_types=1);

$items = ["pen", "book", "lamp"];
$position = array_search("pen", $items, true);

if ($position) {
    echo "Removing " . $items[$position] . PHP_EOL;
} else {
    echo "Not in the list." . PHP_EOL;
}
```

28 CH 8 · MODIFY

Change this program so that it removes an item chosen by value rather than by position, reports clearly when the item is not there, and leaves the remaining keys in a predictable state. Predict what `count()` and the keys look like after the removal before you run it.

```php
<?php

declare(strict_types=1);

$items = ["pen", "book", "lamp"];

array_splice($items, 1, 1);

echo implode(", ", $items) . PHP_EOL;
echo count($items) . PHP_EOL;
```

29 CH 9 · TRACE

Write the five output lines. Explain why the two booleans disagree, and why one of the two ??= lines changed nothing.

```php
<?php

declare(strict_types=1);

$user = ["name" => "Maya", "role" => null];

echo ($user["nickname"] ?? "none") . PHP_EOL;
var_dump(isset($user["role"]));
var_dump(array_key_exists("role", $user));

$user["role"] ??= "member";
$user["name"] ??= "Unknown";

echo $user["role"] . PHP_EOL;
echo $user["name"] . PHP_EOL;
```

30 CH 9 · PREDICT

Predict the three output lines. Say what the second one proves about `usort()`.

```php
<?php

declare(strict_types=1);

$products = [
    ["name" => "Pen", "price" => 1.25],
    ["name" => "Lamp", "price" => 18.00],
    ["name" => "Notebook", "price" => 4.50]
];

$byPrice = $products;
usort($byPrice, fn (array $a, array $b): int => $b["price"] <=> $a["price"]);

echo $byPrice[0]["name"] . PHP_EOL;
echo $products[0]["name"] . PHP_EOL;
echo count($byPrice) . PHP_EOL;
```

31 CH 9 · DEBUG

This total is wrong, and PHP does mention it. Say what is printed, including the warning, and repair the program in the two ways the shape of your data might call for.

```php
<?php

declare(strict_types=1);

$products = [
    ["name" => "Pen", "price" => 1.25],
    ["name" => "Sticker"],
    ["name" => "Lamp", "price" => 18.00]
];

$total = 0.0;

foreach ($products as $product) {
    $total += $product["price"];
}

echo number_format($total, 2) . PHP_EOL;
```

32 CH 9 · BUILD

Start from an empty file. Build a contact directory as an array of associative arrays, keyed for lookup by the lowercase email address. Write `addContact()`, `findContact()`, and a listing loop. Make `findContact()` handle a missing contact without a warning, and make `addContact()` follow a documented rule when the key is already taken.

33 CH 10 · TRACE

Write all five output lines. The fourth one is the reason this question exists.

```php
<?php

declare(strict_types=1);

$inputs = ["17", "18", "abc", "0", ""];

foreach ($inputs as $input) {
    $valid = filter_var($input, FILTER_VALIDATE_INT);

    if ($valid === false) {
        echo $input . " -> Invalid" . PHP_EOL;
    } elseif ($valid < 18) {
        echo $input . " -> Minor" . PHP_EOL;
    } else {
        echo $input . " -> Adult" . PHP_EOL;
    }
}
```

34 CH 10 · DEBUG

Two separate assumptions in this page are wrong. Name both, say what happens on a first visit and what happens when someone submits `<b>Maya</b>`, and repair it.

```php
<?php

declare(strict_types=1);

$name = $_POST["name"];

echo "<h1>Welcome " . $name . "</h1>";
```

35 CH 10 · MODIFY

This page validates one field with one rule. Add a second field, an email address, with its own rules, and add a maximum length to the name. Keep the errors in one array and keep the redisplayed values escaped. Predict what your page does when both fields are wrong at once.

```php
<?php

declare(strict_types=1);

function h(string $value): string
{
    return htmlspecialchars($value, ENT_QUOTES, "UTF-8");
}

$name = trim($_POST["name"] ?? "");
$errors = [];

if ($name === "") {
    $errors[] = "Name is required.";
}

foreach ($errors as $error) {
    echo "<p>" . h($error) . "</p>" . PHP_EOL;
}
```

36 CH 10 · BUILD

Start from an empty file. Build a single page that shows a GET search form, validates the query as non-empty and no longer than a limit you choose, and redisplays what was searched for, escaped, above the results. Run it with `php -S localhost:8000`, then visit it with no query string, with `?q=lamp`, and with `?q=<b>lamp</b>`.

37 CH 11 · TRACE

Write every output line, including what `var_dump()` reports for each of the three values.

```php
<?php

declare(strict_types=1);

$task = ["title" => "Practice PHP", "done" => false, "score" => 9.5, "tags" => []];

$json = json_encode($task, JSON_THROW_ON_ERROR);
echo $json . PHP_EOL;

$back = json_decode($json, true, 512, JSON_THROW_ON_ERROR);

var_dump($back["done"]);
var_dump($back["score"]);
var_dump($back["tags"]);
```

38 CH 11 · PREDICT

Predict the whole output. Say what happened to the first two writes.

```php
<?php

declare(strict_types=1);

$path = __DIR__ . "/notes.txt";

file_put_contents($path, "First" . PHP_EOL);
file_put_contents($path, "Second" . PHP_EOL, FILE_APPEND);
file_put_contents($path, "Third" . PHP_EOL);

$contents = file_get_contents($path);

echo $contents;
echo strlen($contents) . PHP_EOL;

unlink($path);
```

39 CH 11 · DEBUG

Run this one under `php -S localhost:8000` rather than from the terminal, because the defect concerns HTTP headers. Say what warning appears, explain the ordering rule behind it, and repair it.

```php
<?php

declare(strict_types=1);

echo "<p>Starting</p>";
session_start();
$_SESSION["name"] = "Maya";

echo "<p>Stored</p>";
```

40 CH 11 · BUILD

Start from an empty file. Write `loadTasks(string $path): array` and `saveTasks(string $path, array $tasks): void` for a JSON data file. Make `loadTasks()` return an empty array when the file does not exist yet, and make it report malformed JSON without replacing the file. Test all three cases: a missing file, a good file, and a file you have deliberately corrupted.

41 CH 12 · TRACE

Write every visible line in order. Explain why one of the `echo` statements never runs and why the last line does.

```php
<?php

declare(strict_types=1);

function requirePositive(int $value): int
{
    if ($value <= 0) {
        throw new InvalidArgumentException("positive required");
    }

    return $value;
}

try {
    echo requirePositive(3) . PHP_EOL;
    echo requirePositive(0) . PHP_EOL;
    echo "after" . PHP_EOL;
} catch (InvalidArgumentException $error) {
    echo "caught: " . $error->getMessage() . PHP_EOL;
} finally {
    echo "finally" . PHP_EOL;
}
```

42 CH 12 · DEBUG

Nobody registered with that email address, and the lookup finds a real user anyway. Say what the printed SQL looks like, explain what the quotation mark did, and repair it. Then say what the repair does not cover.

```php
<?php

declare(strict_types=1);

$pdo = new PDO("sqlite::memory:");
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$pdo->exec("CREATE TABLE users (id INTEGER PRIMARY KEY, email TEXT NOT NULL)");
$pdo->exec("INSERT INTO users (id, email) VALUES (1, 'maya@example.com')");

$email = "' OR '1'='1";

$sql = "SELECT id, email FROM users WHERE email = '" . $email . "'";

echo $sql . PHP_EOL;

$row = $pdo->query($sql)->fetch(PDO::FETCH_ASSOC);
var_dump($row["email"] ?? null);
```

43 CH 12 · EXPLAIN

Both output lines below came out of the same parameterized insert. Write them, then explain in your own words why one control did not do the other one's job.

```php
<?php

declare(strict_types=1);

$pdo = new PDO("sqlite::memory:");
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$pdo->exec("CREATE TABLE tasks (id INTEGER PRIMARY KEY, title TEXT NOT NULL)");

$stmt = $pdo->prepare("INSERT INTO tasks (title) VALUES (:title)");
$stmt->execute([":title" => "<b>Read Chapter 12</b>"]);

$row = $pdo->query("SELECT title FROM tasks")->fetch(PDO::FETCH_ASSOC);

echo "<li>" . $row["title"] . "</li>" . PHP_EOL;
echo "<li>" . htmlspecialchars($row["title"], ENT_QUOTES, "UTF-8") . "</li>" . PHP_EOL;
```

44 CH 12 · BUILD

Start from two empty files in one folder. Put a pure calculation in `conversions.php`, and in `app.php` load it with `require_once` and use it. Give the function parameter and return types, keep every `echo` out of `conversions.php`, and add a second `require_once` for the same file to see what once means.

45 CH 13 · TRACE

Write the three output lines. Two of them are the same number, and that is the question.

```php
<?php

declare(strict_types=1);

class Player
{
    public function __construct(
        public string $name,
        private int $score = 0
    ) {
    }

    public function addPoints(int $points): void
    {
        if ($points < 0) {
            throw new InvalidArgumentException("Points cannot be negative.");
        }

        $this->score += $points;
    }

    public function score(): int
    {
        return $this->score;
    }
}

$maya = new Player("Maya", 5);
$noah = $maya;
$ava = new Player("Ava");

$noah->addPoints(3);
$ava->addPoints(1);

echo $maya->score() . PHP_EOL;
echo $noah->score() . PHP_EOL;
echo $ava->score() . PHP_EOL;
```

46 CH 13 · DEBUG

This program reports no error, and the value it prints is impossible. Explain what the defect is, then repair the class so that the impossible value cannot be created at all.

```php
<?php

declare(strict_types=1);

class Percentage
{
    public float $value = 0.0;
}

$discount = new Percentage();
$discount->value = 150.0;

var_dump($discount->value);
```

47 CH 13 · BUILD

Start from an empty file. Write a Task class with a title and a private completion state, a `complete()` method, and an `isComplete()` method. Create two tasks, complete one of them, and prove that the other is unaffected. Then say why you did not simply make the completion state public.

48 CH 14 · DEBUG

This is the gradebook average from Project 1. It works, and then it does not. Say what both calls produce, name the acceptance-test row that would have caught it, and repair the function.

```php
<?php

declare(strict_types=1);

function average(array $scores): float
{
    return array_sum($scores) / count($scores);
}

var_dump(average([70, 85, 91]));
var_dump(average([]));
```

49 CH 14 · MODIFY

This is the price-integrity failure from Project 3, reduced to eight lines. Say what it prints, then change it so that the price comes from the server and the request is validated. Predict the new line total before you run it.

```php
<?php

declare(strict_types=1);

$catalog = [
    "notebook" => ["name" => "Notebook", "price" => 4.50],
    "lamp" => ["name" => "Lamp", "price" => 18.00]
];

$submitted = ["id" => "lamp", "price" => "0.01", "quantity" => "2"];

$lineTotal = (float) $submitted["price"] * (int) $submitted["quantity"];

echo number_format($lineTotal, 2) . PHP_EOL;
```

50 CH 14 · BUILD

Start from an empty file. Write `makeSubmission(array $input)`: array for Project 2. It takes the four raw request values, returns a structured record with a generated id and a server timestamp, and throws when the input does not satisfy the rules. Use bin2hex(`random_bytes(8)`) for the id and `date("c")` for the timestamp. Then say why neither of those two values comes from the request.

CHECK YOURSELF AGAINST THESE

Answers

1 CH 1 · PREDICT

The output is:

```
Total:
7
4 + 3
```

The second statement holds an expression, so PHP works out 4 + 3 and sends the value 7. The third holds a double-quoted string, so PHP substitutes the value of each variable and leaves the plus sign as a character between them. Nothing was calculated there. Note also that `echo` takes several values separated by commas, which is why `PHP_EOL` can ride along at the end of each statement.

2 CH 1 · DEBUG

It is a syntax error, so nothing runs at all. PHP reports Parse error: syntax error, unexpected token `"echo"` on line 6, and line 6 is perfectly correct. The defect is on line 5: the assignment has no semicolon at the end. PHP only noticed when the next statement stopped making sense, which is why a parse error so often points one line past the cause. Add the semicolon and the program prints Profile Card.

```php
<?php

declare(strict_types=1);

$title = "Profile Card";
echo $title . PHP_EOL;
```

3 CH 1 · EXPLAIN

This is a logic error. The syntax is valid, so there is no parse error, and every operation succeeds, so there is no runtime failure either. PHP did exactly what was written: multiplication runs before addition, so it worked out 2 * 7 as 14 and then added 3, giving 17. PHP cannot know you meant something else, which is why nothing is reported. Parentheses state the intention, and the program then prints Perimeter: 20. This is the case that makes the P in TPRM worth the effort, because only a prediction catches it.

```php
<?php

declare(strict_types=1);

$width = 7;
$height = 3;
$perimeter = 2 * ($width + $height);

echo "Perimeter: " . $perimeter . PHP_EOL;
```

4 CH 2 · TRACE

The output is:

```
bool(false)
bool(true)
string(4) "1212"
```

The first comparison is strict, and the two values have different types, so it is `false` whatever they look like. The cast in the second line produces an integer, and the comparison then succeeds. The third line concatenates rather than adding, because the dot is the concatenation operator, so the result is four characters of text.

5 CH 2 · PREDICT

The output is:

```
Hello, Maya.
Hello, $name.
Item: Notebook
Cost: $4.50
```

A double-quoted string looks inside itself for variable names and substitutes their values. A single-quoted string hands back exactly the characters you typed, the dollar sign included. The braces on the third line mark where the expression starts and ends, which is what an array element needs. On the last line the backslash escapes the dollar sign, so it is printed as a currency symbol rather than read as the start of a variable name.

6 CH 2 · DEBUG

It prints Next year: 161. The dot is concatenation, not addition, so the character 1 was stuck on the end of the text 16. The value read from a terminal, a form or a file always starts as text, and the repair has two parts. Decide first whether the text is acceptable, with `is_numeric()`, then convert it with (int) and do the arithmetic. The corrected program prints Next year: 17, and a value such as sixteen is now refused rather than producing nonsense.

```php
<?php

declare(strict_types=1);

$ageText = "16";

if (!is_numeric($ageText)) {
    echo "Age must be a number." . PHP_EOL;
} else {
    $nextYear = (int) $ageText + 1;
    echo "Next year: " . $nextYear . PHP_EOL;
}
```

7 CH 3 · TRACE

The variable goes 10, then 15, then 30, then 26. The output is:

```
26
6
2
6.5
```

Read the compound operators out loud: += is add to, *= is multiply by, -= is subtract from. The last three lines are the three ways to divide. Integer division asks how many complete 4s fit into 26, the remainder asks what is left over, and plain division hands back a float.

8 CH 3 · PREDICT

The output is:

```
bool(false)
bool(true)
0.3
0.30
```

A decimal fraction such as 0.1 has no exact binary representation, so the stored sum sits very slightly off 0.3. Strict comparison notices that. The second line asks a different and more useful question: is the gap between the two smaller than a tolerance I chose? Display hides the difference entirely, which is why the third and fourth lines look reassuring and prove nothing.

9 CH 3 · **BUILD**

Every stage gets a name, which is what makes the arithmetic checkable by eye. For 3725 the program prints 1h 2m 5s. For 60 it prints 0h 1m 0s, and for 0 it prints 0h 0m 0s, so the zero boundary needs no special case. The key is keeping the remainder from the hours step in its own variable, since the minutes calculation works on what is left over rather than on the original total.

```php
<?php

declare(strict_types=1);

$totalSeconds = 3725;

$hours = intdiv($totalSeconds, 3600);
$remaining = $totalSeconds % 3600;
$minutes = intdiv($remaining, 60);
$seconds = $remaining % 60;

echo $hours . "h " . $minutes . "m " . $seconds . "s" . PHP_EOL;
```

10 CH 4 · **TRACE**

The output is:

```
95: C
84: C
70: C
59: Needs improvement
```

Nothing is broken in a way PHP can see. The branches are simply in the wrong order. PHP works down the conditions and runs the first `true` one, so the broadest test, `$score` >= 70, catches 95 and 84 before the narrower tests below it are ever considered. The two `elseif` branches are unreachable for any score at all. Test the highest threshold first and the scale works.

11 CH 4 · **PREDICT**

The output is:

```
bool(true)
bool(false)
bool(true)
bool(true)
bool(true)
```

The first comparison is loose, so PHP converts before comparing and calls them equal. The second is strict and notices that one is a string and the other an integer. The third compares two strings of the same type and content. The fourth is the trap worth remembering: the string `"0"` is one of PHP's falsy values, so a loose comparison with `false` succeeds. The last line is strict again, and `"0"` is not the empty string. Any validation that treats a legitimate zero as missing has an example of this defect somewhere in it.

12 CH 4 · DEBUG

It prints Fail. The rule says 60 or more, and the code says more than 60, so exactly one value in the whole range is handled wrongly. The smallest correction is >= in place of >. This is the reason a boundary case belongs in every test set: a score of 59 and a score of 61 both behave correctly here, and only the value sitting on the threshold exposes the defect.

```php
<?php

declare(strict_types=1);

$passMark = 60;
$score = 60;

if ($score >= $passMark) {
    echo "Pass" . PHP_EOL;
} else {
    echo "Fail" . PHP_EOL;
}
```

13 CH 4 · MODIFY

Both versions print International 5-10 days. The match version is shorter because several possibilities can share one arm, and because the whole expression produces a value that you can store, test, or print later rather than a statement that only prints. With the default arm in place, a zone of `"antarctica"` gives Zone not served. Remove the default arm and PHP throws Fatal error: Uncaught UnhandledMatchError: Unhandled match case of type string. For a fixed table of allowed values that refusal is usually what you want, because a silent fallback to some default price is a worse outcome than a stopped program.

```php
<?php

declare(strict_types=1);

$zone = "uk";

$label = match ($zone) {
    "domestic" => "Standard 3-5 days",
    "eu", "uk" => "International 5-10 days",
    default => "Zone not served",
};

echo $label . PHP_EOL;
```

14 CH 5 · TRACE

The output is:

```
3
25
```

The pass for `"8"` gives `$total` 8 and `$counted` 1. The pass for `"abc"` reaches continue and changes nothing. The pass for `"12"` gives 20 and 2. The empty string is not numeric either, so it is skipped as well, which is the one most people miss. The pass for `"5"` gives 25 and 3. The keyword continue abandons the rest of that one pass without ending the loop, which is exactly what you want for input you are turning away rather than input that should stop everything.

15 CH 5 · PREDICT

The output is:

```
checking 0
checking 1
checking 2
found: 2
```

The loop would have run five times. The keyword break leaves it the moment the target is met, so the values 3 and 4 are never reached. Note that the message for 2 is printed before the break, because the `echo` comes first in the body. The starting value of -1 is a deliberate choice: it cannot be confused with the valid index 0, which is a distinction that matters in Chapter 8.

16 CH 5 · DEBUG

It prints:

```
1: book
2: lamp
```

and then a Warning: Undefined array key 3, followed by a line reading 3: with nothing after it. There are two defects and they are easy to confuse. The loop starts at 1, so the element at key 0 is never shown, and it runs while `$i` is less than or equal to the count, so the last pass asks for key 3, which does not exist. For three elements the valid keys are 0, 1 and 2. Start at 0 and use < `count($items)`. If the numbering in the output should start at 1, add one to the index at display time rather than to the loop.

```php
<?php

declare(strict_types=1);

$items = ["pen", "book", "lamp"];

for ($i = 0; $i < count($items); $i += 1) {
    echo ($i + 1) . ": " . $items[$i] . PHP_EOL;
}
```

17 CH 5 · **BUILD**

The empty case has to be settled before the loop, not after it, because an average needs a count that is not zero and the lowest and highest have no meaning at all with nothing to compare. Seed the lowest and highest from the first element rather than from 0, or a set of scores that are all above zero will report a lowest of 0. With the scores [72, 88, 91, 64] the program prints count 4, total 315, lowest 64, highest 91, average 78.75. With one score the lowest, the highest, and the average are all that same score.

```php
<?php

declare(strict_types=1);

$scores = [72, 88, 91, 64];

if ($scores === []) {
    echo "No scores were entered." . PHP_EOL;
} else {
    $total = 0;
    $lowest = $scores[0];
    $highest = $scores[0];

    foreach ($scores as $score) {
        $total += $score;

        if ($score < $lowest) {
            $lowest = $score;
        }

        if ($score > $highest) {
            $highest = $score;
        }
    }

    $average = $total / count($scores);

    echo "Count: " . count($scores) . PHP_EOL;
    echo "Total: " . $total . PHP_EOL;
    echo "Lowest: " . $lowest . PHP_EOL;
    echo "Highest: " . $highest . PHP_EOL;
    echo "Average: " . $average . PHP_EOL;
}
```

18 CH 6 · **TRACE**

The output is:

```
Pen x1 = 0.00
Pen x4 = 5.00
Pen x1 = 1.25
```

The first call supplies only the required argument, so both defaults apply and the line total is zero. The second supplies all three by position. The third uses a named argument to skip over the middle parameter entirely, which positional arguments cannot do. Compare the readability of `invoiceLine("Pen", unitPrice: 1.25)` with `invoiceLine("Pen", 1, 1.25)` and ask which one you would rather meet again in six months.

19 CH 6 · **DEBUG**

It prints 113 and then NULL. The two facts have the same cause. The function prints a value as a side effect and returns nothing, so there is nothing for the assignment to store, and a function declared void hands back `null`. A calculation function should return its result and leave the display to the caller, because the returned number can then be stored, compared, totalled, or sent to a web page, while a printed one is gone. The rewritten version prints Total: 113.00 and then `float(112.99999999999999)`. That second line is not a defect, and it is worth seeing. The multiplier 1.13 has no exact binary representation, so the raw result carries the tiny gap Chapter 3 described. Keeping the raw value in the return and formatting only at the point of display is exactly what lets you notice that, rather than having it rounded away inside a function you cannot inspect.

```php
<?php

declare(strict_types=1);

function addTax(float $subtotal, float $taxRate = 0.13): float
{
    return $subtotal * (1 + $taxRate);
}

$total = addTax(100.0);

echo "Total: " . number_format($total, 2) . PHP_EOL;
var_dump($total);
```

20 CH 6 · **EXPLAIN**

As written, it prints `bool(true)`. PHP converted the string `"72"` into the integer 72 and said nothing at all. Add the declare line and the same call fails with Fatal error: Uncaught `TypeError`: `isPassing()`: Argument #1 (`$score`) must be of type int, string given. The second behavior is the one you want. Everything arriving from a form, a terminal, or a file is a string, and the question of whether that particular string is an acceptable score belongs to your validation code, at the boundary, where you can report it. A silent conversion moves the failure somewhere else, usually into a total that is wrong by an amount nobody can explain.

```php
<?php

declare(strict_types=1);

function isPassing(int $score): bool
{
    return $score >= 60;
}

var_dump(isPassing("72"));
```

21 CH 6 · BUILD

The middle call is the one that matters. A score of 59 and a score of 61 behave the same way under either >= or >, and only the value sitting exactly on the pass mark tells the two rules apart. The program prints `bool(false)`, `bool(true)`, `bool(true)`. Giving the pass mark a default rather than writing 60 into the comparison means the same function serves a course with a different threshold, and the boundary test still works.

```php
<?php

declare(strict_types=1);

function isPassing(int $score, int $passMark = 60): bool
{
    // A score exactly at the pass mark passes, which the second call proves.
    return $score >= $passMark;
}

var_dump(isPassing(59));
var_dump(isPassing(60));
var_dump(isPassing(61));
```

22 CH 7 · TRACE

The output is:

```
4
3
red | green | blue
```

The outer call to `trim()` removed the spaces at the two ends of the whole string and nothing inside it, so splitting on the comma produced four pieces: "red ", " green ", the empty string between the two commas, and " blue". The loop trims each piece separately and discards the empty one, which leaves three. Splitting and cleaning are two separate steps, and running them in one line is what makes this kind of off-by-one impossible to inspect.

23 CH 7 · DEBUG

It sends Hello Maya & Co. with the markup intact, so a browser reads the bold tags as instructions rather than as text. A visitor who can choose that value can therefore put markup, and eventually a script, into your page. The repair is to escape at the point the value enters HTML, with htmlspecialchars($value, ENT_QUOTES, "UTF-8"), which sends Hello Maya & Co. and displays the angle brackets as characters. What the repair does not do is validate. It never asked whether that name is acceptable, and it makes the value safe for one context only. The same escaped string is not automatically fit for a URL, for SQL, or for a script.

```php
<?php

declare(strict_types=1);

function h(string $value): string
{
    return htmlspecialchars($value, ENT_QUOTES, "UTF-8");
}

$displayName = '<b>Maya</b> & Co.';

echo "<p>Hello " . h($displayName) . "</p>" . PHP_EOL;
```

24 CH 7 · BUILD

For the sentence "Maya makes a map" and the letter a, the case-sensitive count is 5 and the case-insensitive count is 5 as well, because every a in that sentence is already lowercase. Change the sentence to "Ava And Maya" and the two counts separate: 3 case-sensitive and 5 case-insensitive. That gap is the whole point of writing the rule down. Deciding that A and a are the same character is a decision about your data, and a program that makes it silently is a program nobody can predict. Reading the string one byte at a time with square brackets is fine for plain ASCII text, and it is not fine for accented or non-Latin characters, where a character can occupy several bytes.

```php
<?php

declare(strict_types=1);

// Normalization rule: the insensitive count treats A and a as the same letter.
$sentence = "Ava And Maya";
$letter = "a";

$exact = 0;
$insensitive = 0;

$lowerSentence = strtolower($sentence);
$lowerLetter = strtolower($letter);

for ($i = 0; $i < strlen($sentence); $i += 1) {
    if ($sentence[$i] === $letter) {
        $exact += 1;
    }

    if ($lowerSentence[$i] === $lowerLetter) {
        $insensitive += 1;
    }
}

echo "Exact: " . $exact . PHP_EOL;
echo "Insensitive: " . $insensitive . PHP_EOL;
```

25 CH 8 · TRACE

The output is:

```
15.5, 18.5, 22, 30
18.5, 22, 15.5, 30
15.5 30
```

A call to `sort()` changes the array it is given rather than handing back a new one. The line `$sorted = $temperatures` copied the array, so the original kept its order and only the copy was sorted. Note also that 22.0 and 30.0 print as 22 and 30, because `echo` shows the value and not the way it was written in the source. Once an array is sorted, the smallest value is at key 0 and the largest at `count()` minus one, which is a cheaper way to find them than a second pass.

26 CH 8 · PREDICT

The output is:

```
MAYA,NOAH,AVA,SAM
Ava,Sam
4
Sam 3
```

Both `array_map()` and `array_filter()` return a new array and leave the original alone, which is why the count is still 4 on the third line. A call to `array_pop()` is the odd one out: it removes the last element from the array you passed in and returns it, so the count drops to 3. Note that `array_filter()` preserves the original keys, so `$short` holds keys 2 and 3 rather than 0 and 1. That does not show in an `implode()`, and it certainly shows in a for loop over indexes.

27 CH 8 · DEBUG

It prints Not in the list. The search succeeded and returned the key 0, and PHP treats 0 as `false` inside a condition, so a match on the very first element is indistinguishable from no match at all. The correction is to compare the result with !== `false` rather than testing it as a boolean, and the repaired program prints Removing pen. Where a yes or no answer is all you need, `in_array($needle, $haystack, true)` avoids the trap altogether. The same shape of defect turns up with `filter_var()`, which can legitimately return the integer 0.

```php
<?php

declare(strict_types=1);

$items = ["pen", "book", "lamp"];
$position = array_search("pen", $items, true);

if ($position !== false) {
    echo "Removing " . $items[$position] . PHP_EOL;
} else {
    echo "Not in the list." . PHP_EOL;
}
```

28 CH 8 · MODIFY

As written it prints pen, lamp and then 2. The starting program removes by position, so it removes whatever happens to sit at key 1. The modified version searches strictly for the value, checks the result with !== `false`, and removes that key. Use `unset()` and the keys keep their original numbers, leaving a gap where the removed element was; use `array_splice()` or `array_values()` afterwards and the list is renumbered from 0. Either is correct, and you have to know which one you chose, because a later for loop over indexes breaks on a gap and a `foreach` does not. The version below prints pen, lamp and then 2, and reports clearly when the value is absent.

```php
<?php

declare(strict_types=1);

$items = ["pen", "book", "lamp"];
$target = "book";

$position = array_search($target, $items, true);

if ($position === false) {
    echo "No such item: " . $target . PHP_EOL;
} else {
    array_splice($items, $position, 1);
    echo implode(", ", $items) . PHP_EOL;
}

echo count($items) . PHP_EOL;
```

29 CH 9 · **TRACE**

The output is:

```
none
bool(false)
bool(true)
member
Maya
```

The key nickname is genuinely absent, so `??` supplied the fallback. The two booleans disagree because they ask different questions. A call to `isset()` asks whether there is a value here, and a present `null` is not a value, so it answers `false`. A call to `array_key_exists()` asks whether there is a key here, and there is. The operator `??=` assigns only when the current value is missing or `null`, so it filled in the `null` role and left the existing name alone. Where a missing field and a `null` field mean different things in your data, `array_key_exists()` is the only one of the two that can tell them apart.

30 CH 9 · **PREDICT**

The output is:

```
Lamp
Pen
3
```

PHP cannot know which field of a record you want to sort by, so `usort()` takes a comparison function and calls it with two elements at a time. The spaceship operator returns a negative number, zero, or a positive one, and putting `$b` first reverses the order, which is why the most expensive product comes out at key 0. The second line proves that `$byPrice` = `$products` copied the array: the caller's order is untouched, because `usort()` sorted the copy. Sorting is a mutation, so make the copy deliberately whenever the original order still matters.

31 CH 9 · **DEBUG**

PHP reports Warning: Undefined array key `"price"` for the middle record, treats that value as `null`, adds nothing, and prints 19.25. The number looks entirely reasonable, which is what makes this worth catching. There are two honest repairs and you have to choose between them. If a record without a price is acceptable, say so and read the field with `$product["price"] ?? 0.0`, which prints the same 19.25 and no longer warns. If a record without a price is broken data, validate the shape and report it instead, which is what the version below does. Silencing the warning without making that decision is the one option that is always wrong.

```php
<?php

declare(strict_types=1);

$products = [
    ["name" => "Pen", "price" => 1.25],
    ["name" => "Sticker"],
    ["name" => "Lamp", "price" => 18.00]
];

$total = 0.0;
$errors = [];

foreach ($products as $product) {
    if (!array_key_exists("price", $product)) {
        $errors[] = "No price for " . $product["name"] . ".";
        continue;
    }

    $total += $product["price"];
}

foreach ($errors as $error) {
    echo $error . PHP_EOL;
}

echo number_format($total, 2) . PHP_EOL;
```

32 CH 9 · **BUILD**

Normalizing the key with `strtolower()` is the decision that makes the lookup reliable, and it is a rule worth writing in a comment, because Maya@Example.com and maya@`example.com` are the same person and two different array keys. The lookup returns `null` rather than warning, which lets the caller decide what an absent contact means. The documented rule here is that adding a duplicate key replaces nothing and reports instead, which is one of the two defensible choices; replacing the existing record is the other, as long as you say so. The program below prints the added contact, the duplicate refusal, the found record, the missing lookup, and then the whole directory.

```php
<?php

declare(strict_types=1);

// Lookup rule: the key is the email address in lowercase.
function addContact(array $directory, string $email, string $name): array
{
    $key = strtolower(trim($email));

    if (array_key_exists($key, $directory)) {
        echo "Already in the directory: " . $key . PHP_EOL;
        return $directory;
    }

    $directory[$key] = ["name" => $name, "email" => trim($email)];

    return $directory;
}

function findContact(array $directory, string $email): ?array
{
    return $directory[strtolower(trim($email))] ?? null;
}

$directory = [];
$directory = addContact($directory, "Maya@Example.com", "Maya");
$directory = addContact($directory, "maya@example.com", "Maya A.");
$directory = addContact($directory, "noah@example.com", "Noah");

$found = findContact($directory, "MAYA@example.com");
echo ($found["name"] ?? "not found") . PHP_EOL;
echo (findContact($directory, "ava@example.com")["name"] ?? "not found") . PHP_EOL;

foreach ($directory as $key => $contact) {
    echo $key . ": " . $contact["name"] . PHP_EOL;
}
```

33 CH 10 · **TRACE**

The output is:

```
17 -> Minor
18 -> Adult
abc -> Invalid
0 -> Minor
 -> Invalid
```

The empty string produces a line that begins with nothing, because the input itself is empty. The line that matters is the fourth. A call to `filter_var()` with FILTER_VALIDATE_INT hands back the integer 0 for the text `"0"`, and 0 is falsy. Written as if (`!$valid`) this program would call a perfectly valid zero invalid. Written as if (`$valid === false`) it accepts it. Any validator that can return a legitimate zero has to be tested with `=== false`.

34 CH 10 · **DEBUG**

The first assumption is that the key is there. On a first visit nobody has submitted anything, so PHP reports Warning: Undefined array key "name" and carries on with null. The second assumption is that the value is text you can trust. Submit Maya and the page sends <h1>Welcome Maya</h1>, and the browser reads those tags as markup. The repair is three steps in order: read the key with an intentional fallback, validate the value against a rule you can state, and escape it at the moment it enters HTML. The repaired page prints <h1>Welcome Maya</h1> for that submission, and Name is required. when the field is empty or absent.

```php
<?php

declare(strict_types=1);

function h(string $value): string
{
    return htmlspecialchars($value, ENT_QUOTES, "UTF-8");
}

$name = trim($_POST["name"] ?? "");
$errors = [];

if ($name === "") {
    $errors[] = "Name is required.";
} elseif (mb_strlen($name) > 60) {
    $errors[] = "Name must be 60 characters or fewer.";
}

if ($errors === []) {
    echo "<h1>Welcome " . h($name) . "</h1>";
} else {
    foreach ($errors as $error) {
        echo "<p>" . h($error) . "</p>";
    }
}
```

35 CH 10 · **MODIFY**

With both fields wrong, a page built this way reports both problems at once rather than stopping at the first. That is the reason the errors go into an array even when there is only one rule: the shape is ready to grow, and a visitor who has three things to correct should be told all three. Note the order of the email checks. Required comes first, then format, because reporting that an empty field is not a valid email address is technically `true` and useless. With an empty name and the value not-an-email submitted, the modified page prints Name is required. and then Email is not valid.

```php
<?php

declare(strict_types=1);

function h(string $value): string
{
    return htmlspecialchars($value, ENT_QUOTES, "UTF-8");
}

$name = trim($_POST["name"] ?? "");
$email = trim($_POST["email"] ?? "");
$errors = [];

if ($name === "") {
    $errors[] = "Name is required.";
} elseif (mb_strlen($name) > 60) {
    $errors[] = "Name must be 60 characters or fewer.";
}

if ($email === "") {
    $errors[] = "Email is required.";
} elseif (filter_var($email, FILTER_VALIDATE_EMAIL) === false) {
    $errors[] = "Email is not valid.";
}

foreach ($errors as $error) {
    echo "<p>" . h($error) . "</p>" . PHP_EOL;
}

if ($errors === []) {
    echo "<p>Thank you, " . h($name) . ".</p>" . PHP_EOL;
}
```

36 CH 10 · **BUILD**

With no query string at all the page shows the form and no error, because a first visit has not searched for anything and should not be told off for it. With ?q=lamp it echoes the term and reports the matches. With ?q=lamp it shows the angle brackets as characters, which is the whole test. Note that the term appears in two output contexts here, once as page text and once in the value attribute of the input, and both need escaping. A GET form is the right choice for a search, because the query belongs in the URL where it can be shared and bookmarked. It would be the wrong choice for anything that changes state on the server.

```php
<?php

declare(strict_types=1);

function h(string $value): string
{
    return htmlspecialchars($value, ENT_QUOTES, "UTF-8");
}

$catalog = ["Lamp", "Notebook", "Pen", "Desk lamp"];

$query = trim($_GET["q"] ?? "");
$errors = [];
$results = [];

if (array_key_exists("q", $_GET)) {
    if ($query === "") {
        $errors[] = "Enter something to search for.";
    } elseif (mb_strlen($query) > 40) {
        $errors[] = "Search terms are limited to 40 characters.";
    } else {
        foreach ($catalog as $product) {
            if (stripos($product, $query) !== false) {
                $results[] = $product;
            }
        }
    }
}
?>
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Search</title>
</head>
<body>
<form method="get">
  <label for="q">Search</label>
  <input id="q" name="q" type="text" value="<?= h($query) ?>">
  <button type="submit">Search</button>
</form>
<?php foreach ($errors as $error): ?>
  <p class="error"><?= h($error) ?></p>
<?php endforeach; ?>
<?php if ($query !== "" && $errors === []): ?>
  <p>Results for <?= h($query) ?>: <?= count($results) ?></p>
  <ul>
  <?php foreach ($results as $product): ?>
    <li><?= h($product) ?></li>
  <?php endforeach; ?>
  </ul>
<?php endif; ?>
</body>
</html>
```

37 CH 11 · TRACE

The output is:

```
{"title":"Practice PHP","done":false,"score":9.5,"tags":[]}
bool(false)
float(9.5)
array(0) {
}
```

The round trip preserved the boolean as a boolean and the number as a number, which is the reason to use JSON rather than inventing a text format. Two details are worth noticing. The empty PHP array encoded as an empty JSON array, so a structure you meant as an empty object comes back as a list. And passing `true` as the second argument to `json_decode()` is what asks for associative arrays; leave it out and `$back["done"]` is a property of an object rather than an array key.

38 CH 11 · PREDICT

The output is:

```
Third
6
```

By default `file_put_contents()` replaces the whole file. The first write created it, the second appended to it because of the FILE_APPEND flag, and the third replaced everything with one line. Six is the five characters of Third plus the line ending. This is the defect that quietly deletes a data file: a save function that meant to add a record and left the flag off. Run the program twice and the result is identical, which is the other half of the lesson, because a destructive write leaves no evidence behind.

39 CH 11 · DEBUG

PHP reports Warning: `session_start()`: Session cannot be started after headers have already been sent, and the session value is not kept. A response has headers first and a body second. The default session mechanism needs to send a cookie, and a cookie is a header, so once the first byte of the body has gone out it is too late to add one. The repair is to move `session_start()` above every `echo`. Two things count as output and catch people out: a blank line or a space before the opening <?php tag, and a stray ?> at the end of an included file followed by a newline. That is the reason a PHP-only file leaves the closing tag off.

```php
<?php

declare(strict_types=1);

session_start();

$_SESSION["name"] = "Maya";

echo "<p>Starting</p>";
echo "<p>Stored</p>";
```

40 CH 11 · **BUILD**

The missing file is the case a beginner program always gets wrong, because the first run of any program that saves data starts without a file. Returning an empty array is honest: there are no tasks yet. The malformed file is different in kind, and the difference matters. An unreadable file is not the same as an empty one, and quietly returning an empty array would let the next save overwrite data that might still be recoverable. Catching JsonException, reporting it, and refusing to continue is the safer choice. Note that both calls carry JSON_THROW_ON_ERROR, and that the path is built with `__DIR__` so the working directory cannot change which file is read.

```php
<?php

declare(strict_types=1);

function loadTasks(string $path): array
{
    if (!file_exists($path)) {
        return [];
    }

    $text = file_get_contents($path);

    if ($text === false) {
        throw new RuntimeException("Could not read " . $path);
    }

    $decoded = json_decode($text, true, 512, JSON_THROW_ON_ERROR);

    if (!is_array($decoded)) {
        throw new RuntimeException("Task file does not hold a list.");
    }

    return $decoded;
}

function saveTasks(string $path, array $tasks): void
{
    $json = json_encode($tasks, JSON_PRETTY_PRINT | JSON_THROW_ON_ERROR);

    if (file_put_contents($path, $json) === false) {
        throw new RuntimeException("Could not write " . $path);
    }
}

$path = __DIR__ . "/tasks.json";

echo count(loadTasks($path)) . PHP_EOL;

saveTasks($path, [["title" => "Practice PHP", "done" => false]]);
echo count(loadTasks($path)) . PHP_EOL;

file_put_contents($path, "{oops}");

try {
    loadTasks($path);
} catch (JsonException $error) {
    echo "Task file is unreadable: " . $error->getMessage() . PHP_EOL;
}

unlink($path);
```

41 CH 12 · TRACE

The output is:

```
3
caught: positive required
finally
```

The first call returns normally. The second throws, and a throw abandons the rest of the try block on the spot, so the word after is never printed. Control moves to the catch block, which matches the type that was thrown. The finally block then runs, and it would have run whether or not anything was thrown, which is what makes it the right place for cleanup. This is the difference between an exception and returning `false`: a caller can ignore a `false` and carry on with a wrong value, and it has to do something about an exception.

42 CH 12 · DEBUG

The printed statement reads SELECT id, email FROM users WHERE email = '' OR '1'='1', and the lookup returns `string(16) "maya@example.com"`. The quotation mark inside the value closed the string literal early, and everything after it stopped being data and became part of the query. The condition OR '1'='1' is `true` for every row. The repair is a prepared statement with a :email placeholder, which sends the structure and the value separately, so the same input finds nothing at all and `var_dump()` reports NULL. What the repair does not cover is output. An email address or a name that came back from a perfectly parameterized query still needs `htmlspecialchars()` when it is rendered into a page, because those are two different boundaries with two different controls.

```php
<?php

declare(strict_types=1);

$pdo = new PDO("sqlite::memory:");
$pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
$pdo->exec("CREATE TABLE users (id INTEGER PRIMARY KEY, email TEXT NOT NULL)");
$pdo->exec("INSERT INTO users (id, email) VALUES (1, 'maya@example.com')");

$email = "' OR '1'='1";

$stmt = $pdo->prepare("SELECT id, email FROM users WHERE email = :email");
$stmt->execute([":email" => $email]);

$row = $stmt->fetch(PDO::FETCH_ASSOC);
var_dump($row["email"] ?? null);
```

43 CH 12 · EXPLAIN

The output is:

```
<li><b>Read Chapter 12</b></li>
<li>&lt;b&gt;Read Chapter 12&lt;/b&gt;</li>
```

The prepared statement did its job perfectly. The title went into the database as data, the angle brackets never touched the structure of the query, and it came back out exactly as it went in. That is the point: a prepared statement protects the database, and it deliberately does not alter your value. Storing what the user meant is correct behavior. The first output line is then unsafe, because nothing escaped that stored value on its way into HTML. Two boundaries, two controls, and neither one is a substitute for the other. Escaping on the way into the database instead would be worse still, because the stored data would carry presentation artifacts everywhere it went.

44 CH 12 · BUILD

Running `php app.php` prints 6.21371. The magic constant `__DIR__` is the folder the file itself sits in, so the path holds up whichever folder you run the command from, which a plain `"conversions.php"` does not. The second `require_once` does nothing at all, and that is what once means: the file is loaded on the first call and skipped on every later one. Use require instead of `require_once` for the second call and PHP stops with Fatal error: Cannot redeclare function `kmToMiles()`, which is the failure `require_once` exists to prevent as a program grows. Keeping `conversions.php` free of output is what lets the same function serve a terminal program, a web page, and a test.

```
// conversions.php
<?php

declare(strict_types=1);

function kmToMiles(float $km): float
{
    return $km * 0.621371;
}

// app.php
<?php

declare(strict_types=1);

require_once __DIR__ . "/conversions.php";
require_once __DIR__ . "/conversions.php";

echo kmToMiles(10) . PHP_EOL;
```

45 CH 13 · TRACE

The output is:

```
8
8
1
```

The line `$noah = $maya` did not create a second player. In PHP an object variable holds a reference-like value, so the assignment left both names pointing at the same object, and adding points through one of them is visible through the other. The object made with new `Player("Ava")` is genuinely separate and keeps its own state. Use clone where you want an independent copy, and be sure you want one. This is the same distinction as the array copy in Chapter 8, with the opposite default: an array is copied on assignment and an object is not.

46 CH 13 · **DEBUG**

It prints `float(150)`, and a discount of 150 percent is not a thing. The defect is the public property: it lets any code anywhere assign any float, so the rule that a percentage lies between 0 and 100 exists only in your head. Making the property private and validating in the constructor moves that rule into the class, where it is enforced once and cannot be forgotten at a call site. The repaired version prints `float(15)` for a valid discount and throws Uncaught `InvalidArgumentException`: Percentage must be between 0 and 100. for the invalid one. Marking the property `readonly` instead would stop later changes without checking the value, so the two techniques answer different questions and are often used together.

```php
<?php

declare(strict_types=1);

class Percentage
{
    public function __construct(private float $value)
    {
        if ($value < 0 || $value > 100) {
            throw new InvalidArgumentException(
                "Percentage must be between 0 and 100."
            );
        }
    }

    public function value(): float
    {
        return $this->value;
    }
}

$discount = new Percentage(15);
var_dump($discount->value());

try {
    new Percentage(150);
} catch (InvalidArgumentException $error) {
    echo "Rejected: " . $error->getMessage() . PHP_EOL;
}
```

47 CH 13 · **BUILD**

The program prints Practice PHP: yes and then Test JSON: no, which is the proof that the two objects hold independent state. Keeping the flag private buys two things. It gives you one place where completion can change, so a rule such as an already-completed task cannot be completed twice has somewhere to live, and it means the outside world talks about the operation rather than the storage. A public property would let any line anywhere set the flag, and the day you need to record a completion time as well, every one of those lines is wrong. Marking the title `readonly` says something further: a task's title is settled when it is created.

```php
<?php

declare(strict_types=1);

final class Task
{
    public function __construct(
        public readonly string $title,
        private bool $done = false
    ) {
        if (trim($title) === "") {
            throw new InvalidArgumentException("A task needs a title.");
        }
    }

    public function complete(): void
    {
        $this->done = true;
    }

    public function isComplete(): bool
    {
        return $this->done;
    }
}

$first = new Task("Practice PHP");
$second = new Task("Test JSON");

$first->complete();

echo $first->title . ": " . ($first->isComplete() ? "yes" : "no") . PHP_EOL;
echo $second->title . ": " . ($second->isComplete() ? "yes" : "no") . PHP_EOL;
```

48 CH 14 · **DEBUG**

The first call prints `float(82)`. The second stops the program with Fatal error: Uncaught DivisionByZeroError: Division by zero. The acceptance-test row is the Empty one: a student with no scores does not cause division by zero. It is worth noticing that this defect is invisible until a student exists who has not been graded yet, which is the normal state of every student on the first day. Return `null` for the empty case and widen the return type to ?float, which makes the caller decide how to display a student with no average. Returning 0.0 instead is defensible only if a student with no scores really does have an average of zero, and in a gradebook that is a different and worse claim.

```php
<?php

declare(strict_types=1);

function average(array $scores): ?float
{
    if ($scores === []) {
        return null;
    }

    return array_sum($scores) / count($scores);
}

var_dump(average([70, 85, 91]));
var_dump(average([]));

$student = ["name" => "Maya", "scores" => []];
$result = average($student["scores"]);

echo $student["name"] . ": " . ($result === null ? "no scores yet" : number_format($result, 1))
    ↳ . PHP_EOL;
```

49 CH 14 · **MODIFY**

As written it prints 0.02, because the browser was allowed to say what a lamp costs. The corrected version prints 36.00. The rule is short and it decides the whole design: a product identifier can be a request, and a price cannot. The client is allowed to say which product, and the server decides what that product costs. Notice what else the correction has to do. The identifier is looked up rather than trusted, so an unknown id is rejected instead of producing a total of zero, and the quantity is validated as a positive integer, since a quantity of -1 turns a purchase into a refund. Test it by editing the submitted price to 0.01 and confirming that the total does not move.

```php
<?php

declare(strict_types=1);

$catalog = [
    "notebook" => ["name" => "Notebook", "price" => 4.50],
    "lamp" => ["name" => "Lamp", "price" => 18.00]
];

$submitted = ["id" => "lamp", "price" => "0.01", "quantity" => "2"];

$id = trim($submitted["id"] ?? "");
$quantity = filter_var($submitted["quantity"] ?? "", FILTER_VALIDATE_INT);

if (!array_key_exists($id, $catalog)) {
    echo "Unknown product." . PHP_EOL;
} elseif ($quantity === false || $quantity < 1) {
    echo "Quantity must be a whole number of at least 1." . PHP_EOL;
} else {
    $product = $catalog[$id];
    $lineTotal = $product["price"] * $quantity;

    echo $product["name"] . " x" . $quantity . " = " . number_format($lineTotal, 2) . PHP_EOL;
}
```

50 CH 14 · **BUILD**

The function returns a record with six fields: id, `created_at`, name, email, subject, and message. Run it twice and the id is different each time, while the record shape is identical. Neither the id nor the timestamp comes from the request, and for the same reason the price does not in Project 3. A client that can choose its own record id can overwrite somebody else's record, and a client that can choose its own timestamp can put a submission in the past or the future. Both are server facts. Note also that validation happens before the record is built, so a rejected submission never consumes an id, and that the function throws rather than returning `false`, which forces the page to answer it.

```php
<?php

declare(strict_types=1);

function makeSubmission(array $input): array
{
    $name = trim($input["name"] ?? "");
    $email = trim($input["email"] ?? "");
    $subject = trim($input["subject"] ?? "");
    $message = trim($input["message"] ?? "");

    $errors = [];

    if ($name === "") {
        $errors[] = "Name is required.";
    }

    if (filter_var($email, FILTER_VALIDATE_EMAIL) === false) {
        $errors[] = "Email is not valid.";
    }

    if ($subject === "" || mb_strlen($subject) > 120) {
        $errors[] = "Subject is required and limited to 120 characters.";
    }

    if ($message === "" || mb_strlen($message) > 2000) {
        $errors[] = "Message is required and limited to 2000 characters.";
    }

    if ($errors !== []) {
        throw new InvalidArgumentException(implode(" ", $errors));
    }

    return [
        "id" => bin2hex(random_bytes(8)),
        "created_at" => date("c"),
        "name" => $name,
        "email" => $email,
        "subject" => $subject,
        "message" => $message
    ];
}

$record = makeSubmission([
    "name" => "Maya",
    "email" => "maya@example.com",
    "subject" => "Question about Chapter 14",
    "message" => "How large should a starter project be?"
]);

echo implode(", ", array_keys($record)) . PHP_EOL;
echo strlen($record["id"]) . PHP_EOL;

try {
    makeSubmission(["name" => "", "email" => "not-an-email"]);
} catch (InvalidArgumentException $error) {
    echo "Rejected: " . $error->getMessage() . PHP_EOL;
}
```