Wie funktioniert php Typisierung

Inhaltsverzeichnis

1. Die grundlegenden PHP-Typen

Die wichtigsten Typen sind:

```
string
int
float
bool
array
object
null
```

Beispiele:

```
$name = "Anna";       // string
$age = 25;            // int
$price = 19.99;       // float
$active = true;       // bool
```

```
$tags = ["php", "web"]; // array
$user = new User();   // object
$value = null;        // null
```

PHP ist dabei standardmäßig **dynamisch typisiert**.

Das bedeutet:

```
$value = "Hallo";
$value = 123;
$value = true;
```

Eine Variable kann also zu unterschiedlichen Zeitpunkten unterschiedliche Typen enthalten.

---

2. Typisierung bei Funktionen

Hier wird Typisierung besonders interessant.

```
function add(int $a, int $b): int
{
    return $a + $b;
}
```

Die Typen stehen hier an drei Stellen:

```
       int $a    int $b
          ↓       ↓
function add(int $a, int $b): int
                              ↑
                         Rückgabetyp
```

Also:

- `$a` muss `int` sein
- `$b` muss `int` sein
- die Funktion muss `int` zurückgeben

Beispiel:

```
$result = add(10, 20);

echo $result; // 30
```

---

3. `declare(strict_types=1)`

Das solltest du unbedingt verstehen.

Am Anfang einer PHP-Datei kannst du schreiben:

```
<?php

declare(strict_types=1);
```

Dann:

```
function add(int $a, int $b): int
{
    return $a + $b;
}
```

und:

```
add(10, 20);      // ✅
add("10", 20);    // ❌ TypeError
```

Ohne `strict_types` kann PHP bei bestimmten Funktionsaufrufen versuchen, Werte automatisch umzuwandeln.

Mit:

```
declare(strict_types=1);
```

sagst du:

> Verwende die angegebenen Typen strikt.

Für moderne PHP-Projekte ist `strict_types=1` eine sehr gute Gewohnheit.

---

4. Rückgabetypen

Du kannst nicht nur Parameter, sondern auch den Rückgabewert typisieren.

```
function getName(): string
{
    return "Anna";
}
```

Das hier wäre falsch:

```
function getName(): string
{
    return 123;
}
```

Mit `strict_types=1` führt das zu einem `TypeError`.

---

5. `void`

Manchmal soll eine Funktion **nichts zurückgeben**.

Dafür gibt es `void`.

```
function sayHello(string $name): void
{
    echo "Hallo $name";
}
```

Die Funktion macht etwas, aber sie liefert keinen Wert zurück.

Also:

```
sayHello("Anna");
```

ist sinnvoll.

Aber:

```
$result = sayHello("Anna");
```

`$result` enthält keinen sinnvollen Rückgabewert.

Ein wichtiger Unterschied:

```
function foo(): void
{
    // kein return
}
```

ist etwas anderes als:

```
function foo(): null
{
    return null;
}
```

`void` bedeutet:

> Diese Funktion gibt keinen Wert zurück.

---

6. Nullable Types

Was ist, wenn ein Wert entweder ein `string` **oder `null`** sein darf?

Dafür gibt es:

```
?string
```

Beispiel:

```
function getUserName(?int $userId): ?string
{
    if ($userId === null) {
        return null;
    }

    return "Anna";
}
```

Das bedeutet:

```
?string
   ↓
string oder null
```

Dasselbe könnte man auch mit einem Union Type schreiben:

```
string|null
```

`?string` ist im Grunde die Kurzschreibweise dafür.

---

7. Union Types

Jetzt kommen wir zu einem wichtigen fortgeschrittenen Konzept.

Ein **Union Type** bedeutet:

> Ein Wert darf Typ A **oder** Typ B sein.

Beispiel:

```
function printId(int|string $id): void
{
    echo $id;
}
```

Jetzt sind beide erlaubt:

```
printId(123);
printId("ABC-123");
```

Aber:

```
printId(true); // ❌
```

Denn `bool` gehört nicht zur Union.

Du kannst auch mehrere Typen kombinieren:

```
function process(int|string|float $value): void
{
    // ...
}
```

Das bedeutet:

```
int
oder
string
oder
float
```

---

8. Union Types mit Logik

Wenn eine Funktion mehrere Typen akzeptiert, musst du häufig unterscheiden, welchen Typ du tatsächlich bekommen hast.

Dafür gibt es beispielsweise `is_string()` und `is_int()`:

```
function formatValue(int|string $value): string
{
    if (is_int($value)) {
        return "ID: " . $value;
    }

    return "Name: " . $value;
}
```

Jetzt:

```
echo formatValue(123);
// ID: 123

echo formatValue("Anna");
// Name: Anna
```

Das nennt man **Type Checking** bzw. **Type Narrowing**.

---

9. `mixed`

Jetzt kommt ein sehr spezieller Typ:

```
mixed
```

`mixed` bedeutet im Grunde:

> Dieser Wert kann praktisch jeden PHP-Typ haben.

Beispiel:

```
function dumpValue(mixed $value): void
{
    var_dump($value);
}
```

Jetzt ist alles erlaubt:

```
dumpValue("Hallo");
dumpValue(123);
dumpValue(12.5);
dumpValue(true);
dumpValue(["a", "b"]);
dumpValue(null);
```

`mixed` entspricht konzeptionell ungefähr:

```
int|string|float|bool|array|object|null|...
```

Es ist also sehr flexibel.

Aber Vorsicht

`mixed` sollte nicht einfach überall verwendet werden.

Schlecht:

```
function calculate(mixed $value): mixed
{
    // ...
}
```

Hier weiß niemand mehr richtig, was die Funktion erwartet oder zurückgibt.

Besser:

```
function calculate(float $price, int $quantity): float
{
    return $price * $quantity;
}
```

Je präziser deine Typen, desto besser.

`mixed` ist vor allem sinnvoll, wenn eine Funktion tatsächlich mit beliebigen Typen arbeiten soll.

---

10. Arrays typisieren

Hier gibt es eine Besonderheit.

Du kannst schreiben:

```
function getTags(): array
{
    return ["php", "mysql", "html"];
}
```

Aber `array` sagt nur:

> Ich bekomme irgendein Array.

Nicht:

> Ich bekomme ein Array aus Strings.

Dafür werden häufig **PHPDoc-Typen** verwendet:

```
/**
 * @return string[]
 */
function getTags(): array
{
    return ["php", "mysql", "html"];
}
```

Damit können IDEs und statische Analysewerkzeuge verstehen:

```
string[]
 ↓
Array mit Strings
```

Man kann auch komplexere Formen beschreiben:

```
/**
 * @param array<string, int> $scores
```

```
 */
function calculateScores(array $scores): int
{
    return array_sum($scores);
}
```

Das bedeutet:

```
Key   → string
Value → int
```

Zum Beispiel:

```
$scores = [
    "Anna" => 90,
    "Max" => 80,
];
```

---

11. Klassen als Typen

Jetzt wird Typisierung richtig mächtig.

Angenommen:

```
class User
{
    public function __construct(
        public string $name
    ) {}
}
```

Dann können wir schreiben:

```
function greet(User $user): string
{
    return "Hallo " . $user->name;
}
```

Jetzt erwartet die Funktion **ein User-Objekt**.

```
$user = new User("Anna");

echo greet($user);
```

Aber:

```
greet("Anna"); // ❌
```

Das ist enorm nützlich, weil du damit ausdrücken kannst:

> Diese Funktion arbeitet ausschließlich mit `User`-Objekten.

---

12. Interfaces

Jetzt kommen wir zu einem der wichtigsten Konzepte in objektorientiertem PHP.

Ein Interface beschreibt einen **Vertrag**.

Beispiel:

```
interface PaymentMethod
{
    public function pay(float $amount): bool;
}
```

Das bedeutet:

> Jede Klasse, die `PaymentMethod` implementiert, muss eine `pay()`-Methode besitzen, die `float` akzeptiert und `bool` zurückgibt.

Zum Beispiel:

```
class CreditCardPayment implements PaymentMethod
{
    public function pay(float $amount): bool
    {
        echo "Bezahle $amount € per Kreditkarte";

        return true;
    }
}
```

Und:

```
class PaypalPayment implements PaymentMethod
{
    public function pay(float $amount): bool
    {
        echo "Bezahle $amount € mit PayPal";

        return true;
    }
}
```

Jetzt kommt der interessante Teil:

```
function checkout(PaymentMethod $payment): void
{
    $payment->pay(100);
}
```

Du kannst jetzt beide Klassen übergeben:

```
checkout(new CreditCardPayment());
checkout(new PaypalPayment());
```

Warum funktioniert das?

Weil beide Klassen den Vertrag

`PaymentMethod`

erfüllen.

Das ist **Polymorphie**.

---

13. Warum Interfaces so wichtig sind

Stell dir einen Online-Shop vor.

Du möchtest nicht:

```
function checkout(CreditCardPayment $payment)
```

weil du dann nur Kreditkarte akzeptierst.

Stattdessen:

```
function checkout(PaymentMethod $payment)
```

Damit ist der Code unabhängig von der konkreten Implementierung.

Heute:

```
CreditCardPayment
```

Morgen:

```
PaypalPayment
```

Übermorgen:

```
BitcoinPayment
```

Solange die Klasse

```
implements PaymentMethod
```

verwendet, funktioniert sie.

Das ist ein zentraler Bestandteil von **Dependency Inversion** und generell sauberem OOP-Design.

---

14. Intersection Types

Jetzt wird es etwas fortgeschrittener.

Ein Union Type bedeutet:

```
A|B
```

also:

> A **oder** B

Ein Intersection Type bedeutet:

A&B

also:

A **und** B

Beispiel:

```
interface Loggable
{
    public function log(): void;
}

interface Cacheable
{
    public function cache(): void;
}
```

Eine Klasse:

```
class User implements Loggable, Cacheable
{
    public function log(): void
    {
        echo "Logging";
    }

    public function cache(): void
    {
        echo "Caching";
    }
}
```

Jetzt können wir schreiben:

```
function process(Loggable&Cacheable $object): void
{
    $object->log();
    $object->cache();
}
```

Das bedeutet:

```
Loggable
    UND
Cacheable
```

Das Objekt muss **beide Interfaces** erfüllen.

```
process(new User()); // ✅
```

Eine Klasse, die nur `Loggable` implementiert:

```
class Logger implements Loggable
{
    public function log(): void {}
}
```

kann nicht übergeben werden:

```
process(new Logger()); // ❌
```

weil `Cacheable` fehlt.

---

15. Union vs. Intersection

Das ist ein Punkt, den man sich gut merken sollte:

Union

`A|B`

bedeutet:

> A **oder** B

Beispiel:

```
function foo(Cat|Dog $animal): void
```

Akzeptiert:

```
Cat
Dog
```

Intersection

`A&B`

bedeutet:

> A **und** B

Beispiel:

```
function foo(Loggable&Cacheable $object): void
```

Akzeptiert nur Objekte, die **beides** sind.

---

16. `never`

`never` ist ein sehr spezieller Rückgabetyp.

Er bedeutet:

> Diese Funktion wird **niemals normal zurückkehren**.

Zum Beispiel eine Funktion, die das Programm beendet:

```
function abort(string $message): never
{
    throw new RuntimeException($message);
```

```
}
```

Danach geht die normale Programmausführung nicht weiter.

```
function getUser(): User
{
    if (!userExists()) {
        abort("User nicht gefunden");
    }

    return new User("Anna");
}
```

PHP bzw. statische Analysewerkzeuge können verstehen:

```
abort()
   ↓
kehrt niemals zurück
   ↓
deshalb muss hier kein User zurückgegeben werden
```

Ein anderes Beispiel:

```
function redirect(string $url): never
{
    header("Location: $url");
    exit;
}
```

Die Funktion endet entweder durch `exit` oder würde anderweitig nicht normal zurückkehren.

`never` vs. `void`

Das ist wichtig:

```
function logMessage(): void
{
    echo "Hallo";
}
```

Die Funktion **kehrt normal zurück**, nur eben ohne Rückgabewert.

Dagegen:

```
function crash(): never
{
    throw new Exception();
}
```

Die Funktion **kehrt niemals normal zurück**.

---

17. `object`

Es gibt auch den Typ:

```
object
```

Beispiel:

```
function inspect(object $value): void
{
    var_dump($value);
}
```

Das akzeptiert grundsätzlich Objekte.

Aber häufig ist eine konkretere Typisierung besser:

```
function inspect(User $user): void
```

statt:

```
function inspect(object $user): void
```

Denn bei `User` weiß dein Programm wesentlich mehr darüber, was mit dem Objekt gemacht werden kann.

---

18. `self`, `parent` und `static`

Bei Klassen gibt es noch spezielle Typen.

Beispiel:

```
class User
{
    public function save(): self
    {
        // speichern
        return $this;
    }
}
```

`self` bedeutet die aktuelle Klasse.

Man kann dann beispielsweise schreiben:

```
$user->save()->save();
```

`static` ist etwas subtiler und bezieht sich auf die **tatsächlich aufgerufene Klasse**.

Das wird besonders bei Vererbung und Fluent APIs relevant.

---

19. Properties typisieren

Typisierung funktioniert auch bei Klassen-Eigenschaften:

```
class User
{
    public string $name;
    public int $age;
    public bool $active;
```

```
}
```

Dann:

```
$user = new User();

$user->name = "Anna";
$user->age = 25;
$user->active = true;
```

Ein falscher Wert:

```
$user->age = "25";
```

kann mit strikter Typisierung zu einem `TypeError` führen.

---

20. Konstruktoren typisieren

Sehr häufig sieht moderner PHP-Code so aus:

```
class User
{
    public function __construct(
        public string $name,
        public int $age,
        public bool $active
    ) {}
}
```

Dann:

```
$user = new User(
    name: "Anna",
    age: 25,
    active: true
);
```

Das ist eine sehr kompakte Schreibweise.

---

21. `readonly`

In modernen PHP-Versionen kannst du zusätzlich `readonly` verwenden:

```
class User
{
    public function __construct(
        public readonly int $id,
        public string $name
    ) {}
}
```

Danach:

```
$user->id = 123; // ❌
```

Die Property kann nach ihrer Initialisierung nicht mehr verändert werden.

Das ist besonders praktisch für Datenobjekte.

---

22. Eine realistischere Anwendung

Jetzt kombinieren wir mehrere Konzepte.

Wir definieren:

```
interface PaymentMethod
{
    public function pay(float $amount): bool;
}
```

Eine Kreditkarte:

```
class CreditCard implements PaymentMethod
{
    public function pay(float $amount): bool
    {
        echo "Kreditkarte: $amount €";

        return true;
    }
}
```

PayPal:

```
class Paypal implements PaymentMethod
{
    public function pay(float $amount): bool
    {
        echo "PayPal: $amount €";

        return true;
    }
}
```

Unser Checkout:

```
function checkout(
    PaymentMethod $payment,
    float $amount
): bool {
    return $payment->pay($amount);
}
```

Jetzt:

```
checkout(new CreditCard(), 49.99);

checkout(new Paypal(), 49.99);
```

Der Checkout weiß überhaupt nicht, **wie** bezahlt wird.

Er weiß nur:

```
Ich bekomme eine PaymentMethod.
Diese besitzt pay().
```

Das ist einer der großen Vorteile von Typisierung + Interfaces.

---

23. Ein Beispiel mit Union + Interface

Manchmal möchtest du mehrere Möglichkeiten erlauben:

```
function printUserId(int|string $id): void
{
    echo $id;
}
```

Oder bei Objekten:

```
interface Admin {}
interface Customer {}
```

Dann könnte man beispielsweise sagen:

```
function process(Admin|Customer $user): void
{
    // ...
}
```

Das Objekt muss entweder `Admin` **oder** `Customer` erfüllen.

Mit:

```
Admin&Customer
```

müsste es hingegen **beides gleichzeitig** sein.

---

24. Typisierung als „Vertrag“

Eine sehr hilfreiche Denkweise ist:

```
function calculate(float $price, int $quantity): float
```

Lies das nicht nur als Syntax.

Lies es als Vertrag:

> „Gib mir einen `float` und einen `int`, und ich garantiere dir einen `float` zurück.“

Bei einem Interface:

```
interface PaymentMethod
{
    public function pay(float $amount): bool;
}
```

lautet der Vertrag:

> „Jede PaymentMethod kann einen Betrag bezahlen und sagt mir mit `bool`, ob es funktioniert hat.“

Bei `never`:

```
function abort(): never
```

lautet der Vertrag:

> „Diese Funktion wird niemals normal zurückkehren.“

Bei `void`:

```
function log(): void
```

lautet er:

> „Diese Funktion kehrt normal zurück, liefert aber keinen Wert.“

---

25. Die wichtigsten Typen auf einen Blick

Typ	Bedeutung
`int`	Ganzzahl
`float`	Kommazahl
`string`	Text
`bool`	`true` / `false`
`array`	Array
`object`	beliebiges Objekt
`null`	kein Wert
`mixed`	praktisch jeder Typ
`void`	kein Rückgabewert
`never`	kehrt niemals normal zurück
`?string`	`string` oder `null`
`A|B`	A oder B
`A&B`	A und B
`User`	konkretes Objekt der Klasse `User`
`PaymentMethod`	Objekt, das dieses Interface erfüllt
