

PHP's Type System Dissected

Understanding how PHP's type system works

Gina P. Banyard

The PHP Foundation

2025-04-17

About me

The PHP Foundation

- BSc in Pure Mathematics from Imperial College London
- Works on PHP since 2019
- Funded by **The PHP Foundation** since 2022
- Cares about type systems
- Cares about PHP semantics

Mastodon

@Girgias@phpc.social

Website gpb.moe

GitHub [Girgias](https://github.com/Girgias)

PHP's Type System

What is a Type System

A **type system** is a logical system comprising a set of rules that assigns a property called a *type* to every “term”.

A type system dictates the operations that can be performed on a term.

— Wikipedia [1]

Why care about types?

The purpose of abstracting is **not** to be vague, but to create a new semantic level in which one can be absolutely precise.

— E. W. Dijkstra [2]

- Atomic types
 - Primitive types
 - User defined types
 - Singleton types
 - The **callable** type
- Composite types
- Type aliases

Atomic types: Primitive types

Universal Type `mixed` (PHP 8.0[3])

Resource Type

Object Type `object` (PHP 7.2[4])

Hash Table Type `array`

Scalar Type `bool`, `int`, `float`, `string`

Unit Type `null` (PHP 8.0*[5])

Empty Type `never` (PHP 8.1[6])

And a special return only type:

`void`

Atomic types: User defined classes

Also called **class-types**:

Classes *aka product* types

Interfaces

Enumerations *aka sum* (/co-product) types (PHP 8.1[7])

relative-class types:

self

parent

static only as a return type (PHP 8.0[8])

Atomic types: Singleton types

A singleton type is a concrete *subtype* of a type.

false (PHP 8.0*[5])

true (PHP 8.2[9])

Warning

It's impossible to define a value type in user land. Create an enumeration instead.

Atomic types: **callable** type

Represents a function:

- **Closure**
 - anonymous functions
 - obtainable via the `my_function(...)` syntax (PHP 8.1[10])
 - obtainable via `ReflectionFunction::getClosure()`
- A string with the name of the function `"my_function"`
- An object/method name pair: `[$object, "method_name"]`
- Instance of an object that implements `__invoke()`

Warning

It's impossible to define a class property as **callable**.

Composite types

A composite type is a type combining multiple atomic types.

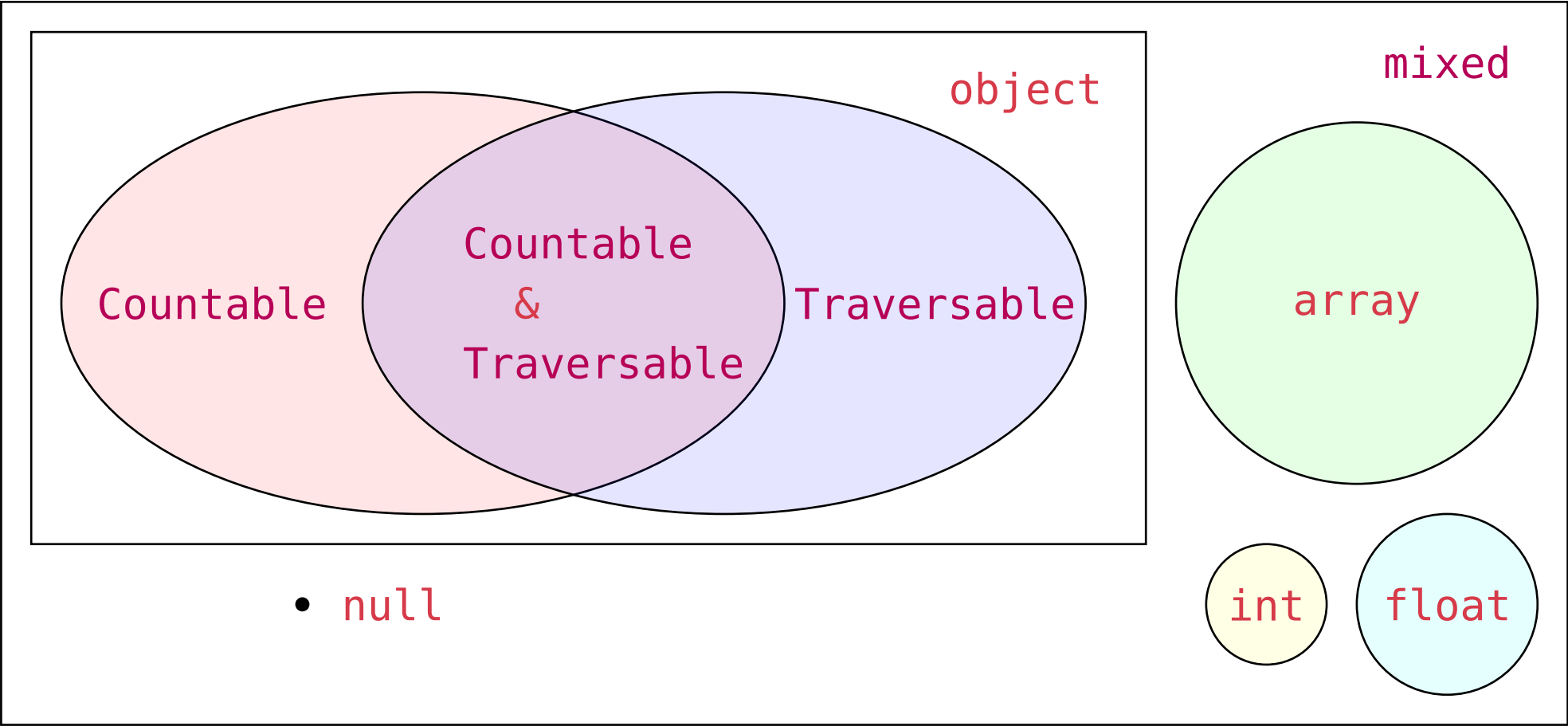
Simple union type (PHP 8.0[11]) $T|S$

Intersection type (PHP 8.1[12]) $X\&Y$

DNF¹ types (PHP 8.2[13]) $(A\&B) | (X\&Y)$

¹In boolean logic, a *Disjunctive Normal Form* or *DNF* is a canonical normal form of a logical formula consisting of a disjunction of conjunctions; it can also be described as an **OR** of **ANDs**.

Composite types: visualization



never

Composite types: why do we care?

Intersection of types allows using *every* API provided by each type

Union of types only allows using the *common* API of all types

As of PHP 8.2, `iterable` is a compile time alias. [14]

`iterable := Traversable|array`

Before it was a *pseudo*-type

Warning

It's impossible to define a type alias in user land.

Internal representation: Values are **zvals**

PHP's Type System

```
1 struct _zval_struct {  
2     zend_value value;  
3     uint32_t type_info;  
4 };
```

```
1 #define IS_NULL      1  
2 #define IS_FALSE     2  
3 #define IS_TRUE      3  
4 #define IS_LONG      4  
5 #define IS_DOUBLE     5  
6 #define IS_STRING     6  
7 #define IS_ARRAY      7  
8 #define IS_OBJECT     8  
9 #define IS_RESOURCE   9  
10 /* ... */
```

Internal representation: zend_type

ptr is either a class name

```
1 typedef struct {  
2     void *ptr;  
3     /* Bit-mask of  
4        primitive  
5        types and type info  
6        */  
7     uint32_t type_mask;  
8 } zend_type;
```

Internal representation: zend_type

ptr is either a class name

or a list of type

```
1 typedef struct {  
2     void *ptr;  
3     /* Bit-mask of  
4        primitive  
5        types and type info  
6        */  
7     uint32_t type_mask;  
8 } zend_type;
```

```
1 typedef struct {  
2     uint32_t num_types;  
3     zend_type types[1];  
4 } zend_type_list;
```

Subtyping and LSP

What is subtyping

In programming language theory, **subtyping** is a form of type polymorphism, in which a subtype is a data type that is related to another data type (the supertype) by some notion of substitutability.

If S is a subtype of T , the subtyping relation (written as $S <: T$, $S \sqsubseteq T$, or $S \leq: T$) means that any term of type S can **safely** be used in **any** context where a term of type T is expected.

— Wikipedia [15]

Liskov Substitution Principle

The Liskov Substitution Principle (**LSP**) is a particular definition of a subtyping relation, called strong behavioural subtyping.

Formulated by Barbara Liskov and Jeannette Wing in 1994:

Let $\varphi(x)$ be a property provable about objects x of type T .

Then $\varphi(y)$ should be *true* for objects y of type S where S is a **subtype** of T .

— B. H. Liskov and J. M. Wing [16]

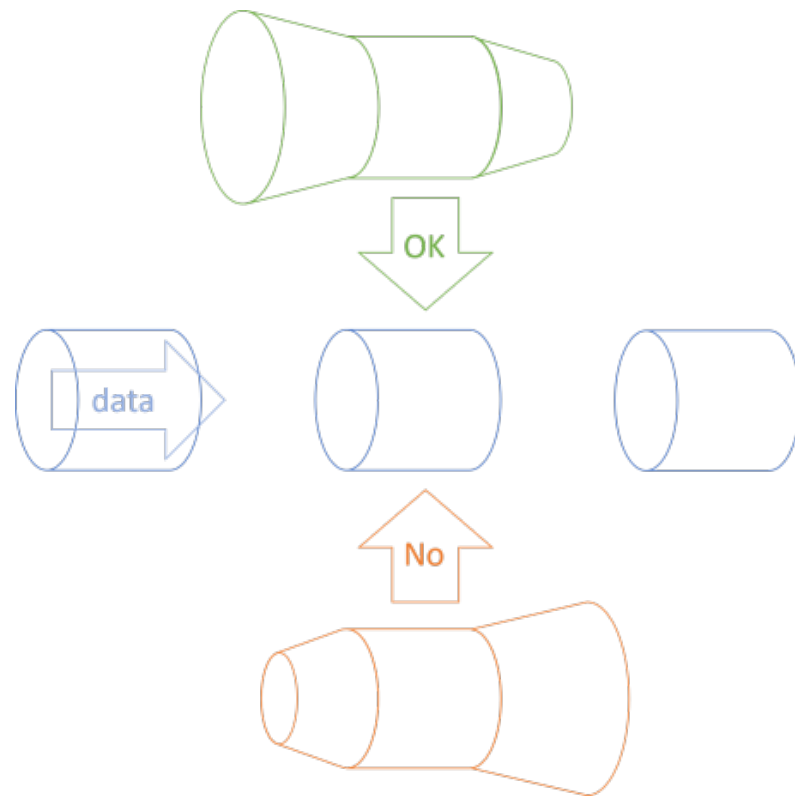


Figure 1: Visualization of Liskov Substitution Principle as a pipe. [17]

- Pre-conditions** cannot be strengthened in the subtype
- Post-conditions** cannot be weakened in the subtype
- Invariants** must be preserved in the subtype
- History Rule** constraints must be preserved in the subtype

Methods cannot add mandatory parameters

Parameter types of methods must be *contra-variant*,
i.e. a supertype

The Return type of methods must be *co-variant*, i.e. a subtype

Property types must be *co and contra-variant*

Expanding on the History Rule

Subtypes must not alter the state of the supertype in ways that are not possible for the supertype to do.

Expanding on the History Rule

Subtypes must not alter the state of the supertype in ways that are not possible for the supertype to do.

Members declared protected are far more open to abuse than members declared private. In particular, declaring data members protected is usually a design error.

— B. Stroustrup [18]

When is $S \sqsubseteq T$?

Generally $S \sqsubseteq T$ if:

- S intersects T with a new type U ($T \& U \sqsubseteq T$)
- S removes a type T_i from a union of types $T := T_1 | T_2 | \dots | T_n$

Examples of $S \sqsubseteq T$: union types

```
1 class Super1 {  
2     public function foo(): T|S|U|V {}  
3 }  
4  
5 class Sub1 extends Super1 {  
6     public function foo(): U|V {}  
7 }
```

Examples of $S \sqsubseteq T$: intersection types

```
1 class Super2 {  
2     public function foo(): A&B {}  
3 }  
4  
5 class Sub2 extends Super2 {  
6     public function foo(): A&B&C&D {}  
7 }
```

Examples of $S \sqsubseteq T$

```
1  interface A {}
2  class X implements A {}
3  class Y implements A {}
4
5  class Super3 {
6      public function foo(): A {}
7  }
8  class Sub3 extends Super3 {
9      public function foo(): X|Y {}
10 }
```

Examples of $S \sqsubseteq T$

```
1  interface A {}
2  interface B {}
3  class X implements A, B {}
4  class Y implements A, B {}
5
6  class Super4 {
7      public function foo(): A&B {}
8  }
9  class Sub4 extends Super4 {
10     public function foo(): X|Y {}
11 }
```

Examples of $S \sqsubseteq T$

```
1  interface A {}
2  interface B {}
3  class X implements A, B {}
4  class Y implements A, B {}
5
6  class Super5 {
7      function foo(): (A&B)|D {}
8  }
9  class Sub5 extends Super5 {
10     function foo(): X|Y|D {}
11 }
```

Examples of $S \sqsubseteq T$

```
1  interface A {}
2  interface B {}
3  interface C {}
4  interface X extends A {}
5
6  class Super6 {
7      public function foo(): A|B {}
8  }
9  class Sub6 extends Super6 {
10     public function foo(): X&C {}
11 }
```

MATH ALERT

$U \sqsubseteq V$ with U a union type

$\forall$ means “For All”, $\exists$ means “There exists”.

Let $U = \{U_1, \dots, U_n\}$ and $V = \{V_1, \dots, V_m\}$ be sets of types.

if V is a union type:

$$U_1 \mid \dots \mid U_n \sqsubseteq V_1 \mid \dots \mid V_m \iff \forall U_i, \exists V_j : U_i \sqsubseteq V_j$$

if V is an intersection type:

$$U_1 \mid \dots \mid U_n \sqsubseteq V_1 \& \dots \& V_m \iff \forall U_i, \forall V_j : U_i \sqsubseteq V_j$$

Basically: iterate over the types U_i of U and verify $U_i \sqsubseteq V$

$U \sqsubseteq V$ with U a union type

$\forall$ means “For All”, $\exists$ means “There exists”.

Let $U = \{U_1, \dots, U_n\}$ and $V = \{V_1, \dots, V_m\}$ be sets of types.

if V is a union type:

$$U_1 \mid \dots \mid U_n \sqsubseteq V_1 \mid \dots \mid V_m \iff \forall U_i, \exists V_j : U_i \sqsubseteq V_j$$

if V is an intersection type:

$$U_1 \mid \dots \mid U_n \sqsubseteq V_1 \& \dots \& V_m \iff \forall U_i, \forall V_j : U_i \sqsubseteq V_j$$



Basically: iterate over the types U_i of U and verify $U_i \sqsubseteq V$

```
1  $u = [$u1, $u2, /* ... */, $uN];
2  $v = [$v1, $v2, /* ... */, $vM];
3  $early_status_exit = false;
4  foreach ($u as $type) {
5      if (is_intersection_type($type)) {
6          $status = is_intersection_subtype($type, $v);
7      } else {
8          $status = is_single_type_subtype($type, $v);
9      }
10     if ($status == $early_exit_status) { return $status; }
11 }
```

$U \sqsubseteq V$ with U an intersection type

$\forall$ means “For All”, $\exists$ means “There exists”.

Let $U = \{U_1, \dots, U_n\}$ and $V = \{V_1, \dots, V_m\}$ be sets of types.

if V is a union type:

$$U_1 \& \dots \& U_n \sqsubseteq V_1 \mid \dots \mid V_m \iff \exists V_j, \exists U_i : U_i \sqsubseteq V_j$$

if V is an intersection type:

$$U_1 \& \dots \& U_n \sqsubseteq V_1 \& \dots \& V_m \iff \forall V_j, \exists U_i : U_i \sqsubseteq V_j$$

$U \sqsubseteq V$ with U an intersection type

$\forall$ means “For All”, $\exists$ means “There exists”.

Let $U = \{U_1, \dots, U_n\}$ and $V = \{V_1, \dots, V_m\}$ be sets of types.

if V is a union type:

$$U_1 \& \dots \& U_n \sqsubseteq V_1 \mid \dots \mid V_m \iff \exists V_j, \exists U_i : U_i \sqsubseteq V_j$$


if V is an intersection type:

$$U_1 \& \dots \& U_n \sqsubseteq V_1 \& \dots \& V_m \iff \forall V_j, \exists U_i : U_i \sqsubseteq V_j$$

Order of quantifiers is *flipped* $\Rightarrow$ need to iterate on V first.

$U \sqsubseteq V$ with U an intersection type

$\forall$ means “For All”, $\exists$ means “There exists”.

Let $U = \{U_1, \dots, U_n\}$ and $V = \{V_1, \dots, V_m\}$ be sets of types.

if V is a union type:

$$U_1 \& \dots \& U_n \sqsubseteq V_1 \mid \dots \mid V_m \iff \exists V_j, \exists U_i : U_i \sqsubseteq V_j$$

if V is an intersection type:

$$U_1 \& \dots \& U_n \sqsubseteq V_1 \& \dots \& V_m \iff \forall V_j, \exists U_i : U_i \sqsubseteq V_j$$



```
1  function is_intersection_subtype($sub, $super) {
2      $early_status = !is_intersection_type($super);
3      foreach ($super as $single) {
4          if (is_intersection_type($single)) {
5              $status = is_intersection_subtype($sub, $single);
6          } else {
7              $status = is_subtype_of_class($sub, $single);
8          }
9          if ($status == $early_status) { return $status; }
10     }
11     return !$early_status; }
```

We explore a new pedagogical framing of the LSP. [...] we advocate an operationalised version of the rule: that a sub-type must pass its supertype's **black box** tests for each of its overriding methods.

[...] black box tests should be written to capture conformance to a specification without overfitting or checking implementation internals.

— E. Baniassad and A. J. Summers [19]

Future of PHP's type system?

User defined type aliases

Future of PHP's type system?

```
typedef numeric int|float
```

```
type numeric as int|float
```

Allowing the use of `int` and `string` values as singleton types

- `1|2|4`
- `"hello"|"world"`
- `SOME_CONSTANT|ANOTHER_CONSTANT`

Specify signature of `callable`/`Closure` parameters/return values

```
foo(fn<int,string>:bool $callable) { /* ... */ }
```

For functions that have the following signature:

```
1 function bar(int $i, string $s): bool {}
```

```
1 class Collection<T> {  
2     private array<T> $stack = [];  
3     public function push(T $v) {  
4         $this->stack[] = $v;  
5     }  
6     public function pop(): T {  
7         return array_pop($this->stack);  
8     }  
9 }
```

```
1  class Collection<T> {  
2      private array<T> $stack = [];  
3      public function push(T $v) {  
4          $this->stack[] = $v;  
5      }  
6      public function pop(): T {  
7          return array_pop($this->stack);  
8      }  
9  }
```

Generics are hard to make fast at run-time with a good DX. [20]

Associated types

Future of PHP's type system?

```
1  interface I {
2      type T: string|int;
3      type V;
4      public function get(T $offset): V;
5      public function set(T $offset, V $value): void;
6  }
7  class C implements I {
8      public function get(string $offset): User { /* ... */ }
9      public function set(string $offset, User $value): void
10     { /* ... */ }
11 }
```

Normal PHP references do not enforce a type constraint:

```
1 function foo(array &$v) {  
2     $v = 5;  
3 }  
4 $a = [];  
5 foo($a);  
6 var_dump($a);  
int(5)
```

Normal PHP references do not enforce a type constraint:

```
1 function foo(inout array $v) {  
2     $v = 5;  
3 }  
4 $a = [];  
5 foo($a);
```

TypeError: inout argument 1 passed to foo() must be of the type array, int assigned on line 2

Communicate what effects a function does

- `function print(string $str): void!output {}`
- `function fopen(string $path): resource!io|warning {}`
- `function div(float $n, float $q):
float!throw<DivisionByZero> {}`
- `function now(): DateTimeImmutable!time`

PHP's `strict_types`

Audience question

PHP's `strict_types`

Who has *heard* about `strict_types`?

Audience question

Who has *heard* about `strict_types`?

Who **knows** what `strict_types` does?

- String
- Integral and String
- Numeric
- Logical
- Comparative
- Function
- “Special” type juggling contexts
 - Increment/Decrement operators
 - Array offsets
 - String offsets
 - `exit()` prior to PHP 8.4

String Type Juggling Context

PHP's `strict_types`

Context used for `echo`, `print`, string interpolation, and `.` operator.

```
1 var_dump(true . " hello");
```

```
string(7) "1 hello"
```

```
2 $a = [1, 2, 3];
```

```
3 echo $a . " are great";
```

```
Warning: Array to string conversion on line 3
```

```
Array are great
```

```
4 print new stdClass();
```

```
Fatal error: Uncaught Error: Object of class stdClass could  
not be converted to string
```

Integral and String Type Juggling Context

PHP's `strict_types`

Context when using bitwise operators (`|`, `&`, `~`, `^`)

```
1 var_dump("123" | "abc");
```

```
string(3) "qrs"
```

```
2 var_dump(36 | "123");
```

```
int(127)
```

```
3 var_dump(36 | gmp_init(25));
```

```
object(GMP)2 (1) { ["num"]=> string(2) "61" }
```

Numeric Type Juggling Context

PHP's `strict_types`

Context when using arithmetic operators (+, -, *, /, **, %)

```
1 var_dump(36 + true);
```

```
int(37)
```

```
2 var_dump("123" + "1e3");
```

```
float(1123)
```

```
3 var_dump(36 + gmp_init(25));
```

```
object(GMP)2 (1) { ["num"]=> string(2) "61" }
```

Logical Type Juggling Context

PHP's `strict_types`

Context when using conditional statements, the ternary operator, or logical operators (`||`, `&&`, `!`, `and`, `or`, `xor`)

```
1 $o = gmp_init(10);  
2 if ($o) {  
3     echo "Hello";  
4 }
```

PHP 8.4 $\geq$: Hello

PHP 8.4 $<$: Recoverable fatal error: Object of class GMP could not be converted to bool on line 2

Comparative Type Juggling Context

PHP's strict_types

Context when using a comparison operator (==, ===, !=, <, <=, >, >=, <=>)

Type OP1	Type OP2	Result
string null	string null	Convert null to "", numerical or lexical comparison
bool null	mixed	Convert both sides to bool: false < true
object	object	Built-in classes can overload comparisons, same classes compare properties, else incomparable
string resource int float	string resource int float	Convert string and resource to number, usual mathematical comparison

Type OP1	Type OP2	Result
array	array	OP with less elements is smaller, if a key of OP1 does not exist in OP2 $\Rightarrow$ uncomparable, else compare elements by value.
object	mixed	object is always greater
array	mixed	array is always greater

Comparison Rules: Implications

`[]` $\Leftrightarrow$ `true` $\Rightarrow OP1 < OP2$

`"1"` $\Leftrightarrow$ `"01"` $\Rightarrow OP1 = OP2$

`"0e5"` $\Leftrightarrow$ `"0e9"` $\Rightarrow OP1 = OP2$

`"hello"` $\Leftrightarrow$ `"world"` $\Rightarrow OP1 < OP2$

Comparison Rules: Implications

`[]` $\Leftrightarrow$ `true` $\Rightarrow OP1 < OP2$

`"1"` $\Leftrightarrow$ `"01"` $\Rightarrow OP1 = OP2$

`"0e5"` $\Leftrightarrow$ `"0e9"` $\Rightarrow OP1 = OP2$

`"hello"` $\Leftrightarrow$ `"world"` $\Rightarrow OP1 < OP2$

Incomparability

```
1 $a1 = [15, 20];  
2 $a2 = ["a" => "a", "b" => "b"];  
3 var_dump($a1 <=> $a2);  
4 var_dump($a2 <=> $a1);  
int(1)  
int(1)
```

Function Type Juggling Context

PHP's `strict_types`

Context for values passed to a typed parameter, property, or returned from a function which declares a return type.

In this context, the value must be a value of the type.

Function Type Juggling Context

PHP's `strict_types`

Context for values passed to a typed parameter, property, or returned from a function which declares a return type.

In this context, the value must be a value of the type.

Exceptions:

- `int` to `float` promotion
- Type and value are scalars, value coerced to type if compatible
- The `string` type accepts objects that are castable to `string`

Warning

Internal functions coerce `null` for scalar type, deprecated in PHP 8.1

```
1 function something_with_float(float $f) {  
2     var_dump($f);  
3 }  
4  
5 something_with_float(15);  
float(15)
```

If multiple scalars types are allowed, the order is the following:

1. `int`
2. `float`
3. `string`
4. `bool`

Scalar Types Coercion: **float** to **int|string** PHP's `strict_types`

```
1 function thing_with_int_or_string(int|string $v) {  
2     var_dump($v);  
3 }
```

```
4  
5 thing_with_int_or_string(15.6);
```

Deprecated: Implicit conversion from float 15.6 to int loses precision on line 5
int(15)

Scalar Types Coercion: **string** to **int|float** PHP's `strict_types`

```
1 function thing_with_int_or_float(int|float $v) {  
2     var_dump($v);  
3 }  
4  
5 thing_with_int_or_float("15.6");  
6 thing_with_int_or_float("25");  
float(15.6)  
int(25)
```

- Array offsets
 - Throw **Error** for **array** and **object** offsets
 - **null** cast to an empty string `""`
 - **resource**, **bool**, **float** cast to **int**
 - Integer **strings** cast to **int** (e.g. `"7"` but **not** `"007"`)
- String offsets
 - Non-**int** values emit an **E_WARNING** or throw an **Error**
- **exit()** prior to PHP 8.4
 - **int** used as status code, everything else cast to **string**
- Increment/Decrement operators
 - Woooooooo boooooooooiiii

Special Type Juggling Contexts: ++/--

PHP's `strict_types`

Behaves like `$v += 1` or `$v -= 1` for:

- `int`
- `float`
- Internal objects that overload `+/ -`

`TypeError` for:

- `array`
- `resource`
- `object`

++/-- and nullPHP $\leq$ 8.2

```
1 $n = null;  
2 var_dump(--$n);  
3 $n = null;  
4 var_dump(++$n);
```

NULL

int(1)

++/-- and nullPHP $\geq$ 8.3

```
1 $n = null;  
2 var_dump(--$n);  
3 $n = null;  
4 var_dump(++$n);
```

Warning: Decrement on type null has no effect, this will change in the next major version of PHP on line 2

NULL

int(1)

`++/--` with `false`PHP $\leq$ 8.2

```
1 $v = false;  
2 var_dump(--$v);  
3 $v = false;  
4 var_dump(++$v);  
bool(false)  
bool(false)
```

++/-- with falsePHP $\geq$ 8.3

```
1 $v = false;  
2 var_dump(--$v);  
3 $v = false;  
4 var_dump(++$v);
```

```
Warning: Decrement on type bool has no effect, ...  
bool(false)
```

```
Warning: Increment on type bool has no effect, ...  
bool(false)
```

++/-- with truePHP $\leq$ 8.2

```
1 $v = true;  
2 var_dump(--$v);  
3 $v = true;  
4 var_dump(++$v);  
bool(true)  
bool(true)
```

++/-- with truePHP $\geq$ 8.3

```
1 $v = true;  
2 var_dump(--$v);  
3 $v = true;  
4 var_dump(++$v);
```

```
Warning: Decrement on type bool has no effect, ...  
bool(true)
```

```
Warning: Increment on type bool has no effect, ...  
bool(true)
```

++/-- with stringPHP $\leq$ 8.2

```
1 $v = "";  
2 var_dump(--$v);  
3 $v = "";  
4 var_dump(++$v);  
int(-1)  
string(1) "1"
```

++/-- with string

PHP $\geq$ 8.3

```
1 $v = "";  
2 var_dump(--$v);  
3 $v = "";  
4 var_dump(++$v);
```

Deprecated: Decrement on empty string is deprecated as non-numeric

int(-1)

Deprecated: Increment on non-alphanumeric string is deprecated

string(1) "1"

```
1 $s = "Z";  
2 var_dump(++$s);  
string(2) "AA"
```

```
3 $s = "Z "; // Trailing space, deprecated as of 8.3  
4 var_dump(++$s);  
string(2) "Z "
```

```
5 $s = " Z"; // Leading space, deprecated as of 8.3  
6 var_dump(++$s);  
string(2) " A"
```

++/-- with **string**: what could go wrong

PHP's `strict_types`

```
1 $s = "4y6";  
2 for ($i = 1; $i < 100; $i++) {  
3     $s++;  
4 }  
5 var_dump($s);
```

++/-- with string: what could go wrong

PHP's `strict_types`

```
1 $s = "4y6";  
2 for ($i = 1; $i < 100; $i++) {  
3     $s++;  
4 }  
5 var_dump($s);  
float(50)
```

WAT

```
1 $s = "5d9";  
2 var_dump(++$s);  
string(3) "5e0"
```

```
3 var_dump(++$s);  
float(6)
```

When does **strict_types** have an effect?

PHP's **strict_types**

- String
- Integral and String
- Numeric
- Logical
- Comparative
- Function
- “Special” type juggling contexts
 - Increment/Decrement operators
 - Array offsets
 - String offsets
 - **exit()** prior to PHP 8.4

When does **strict_types** have an effect?

PHP's **strict_types**

- String
- Integral and String
- Numeric
- Logical
- Comparative
- **Function**
- “Special” type juggling contexts
 - Increment/Decrement operators
 - Array offsets
 - String offsets
 - **exit()** prior to PHP 8.4

What does `strict_types` do?

Only enabled in PHP scripts that use `declare(strict_types=1);`

Disables coercion of scalar types in the following cases **only**:

- Arguments passed to function calls made in userland
- Return value for user defined functions
- Value assignment to a typed property

strict_types does not change binary ops

PHP's strict_types

```
1 declare(strict_types=1);  
2 var_dump(10 + "45");  
3 var_dump(true . " hello");
```

```
int(55)
```

```
string(7) "1 hello"
```

strict_types does not change compare ops

PHP's **strict_types**

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

```
2 var_dump("1" == "01");
```

```
3 var_dump(14 == "014");
```

```
4 var_dump([] > true);
```

```
bool(true)
```

```
bool(true)
```

```
bool(false)
```

strict_types does not change engine calls

PHP's strict_types

```
1 declare(strict_types=1);
2 $is_odd = fn (int $i): bool => ($i % 2);
3 $a = ['1', '2', 3, 4, '5.0', '6.0', true, false];
4 var_dump(array_filter($a, $is_odd));
array(3) {
    [0]=>
    string(1) "1"
    [2]=>
    int(3)
    [4]=>
    string(3) "5.0"
    [6]=>
    bool(true)
}
```

The `strict_types` declare was a *mistake*.

Thank you!

Bibliography

- [1] “Type system.” Apr. 17, 2025. Accessed: Apr. 20, 2025. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Type_system&oldid=1286041555

- [2] E. W. Dijkstra, “The humble programmer,” *Communications of the ACM*, vol. 15, no. 10, pp. 859–866, Oct. 1972, doi: 10.1145/355604.361591.

- [3] M. Kocsis and D. Ackroyd, “PHP RFC: Mixed Type v2.” Accessed: Apr. 20, 2025. [Online]. Available: https://wiki.php.net/rfc/mixed_type_v2

- [4] M. Brzuchalski and D. Ackroyd, “PHP RFC: Object typehint.” Accessed: Apr. 20, 2025. [Online]. Available: <https://wiki.php.net/rfc/object-typehint>
- [5] G. P. Banyard, “PHP RFC: Allow null and false as stand-alone types.” Accessed: Apr. 20, 2025. [Online]. Available: <https://wiki.php.net/rfc/null-false-standalone-types>
- [6] M. Brown and O. Mirtes, “PHP RFC: noreturn type.” Accessed: Apr. 20, 2025. [Online]. Available: https://wiki.php.net/rfc/noreturn_type
- [7] I. Tovilo and L. Garfield, “PHP RFC: Enumerations.” Accessed: Apr. 20, 2025. [Online]. Available: <https://wiki.php.net/rfc/enumerations>

- [8] N. Popov, “PHP RFC: Static return type.” Accessed: Apr. 20, 2025. [Online]. Available: https://wiki.php.net/rfc/static_return_type
- [9] G. P. Banyard, “PHP RFC: Add true type.” Accessed: Apr. 20, 2025. [Online]. Available: <https://wiki.php.net/rfc/true-type>
- [10] N. Popov and J. Watkins, “PHP RFC: First-class callable syntax.” Accessed: Apr. 20, 2025. [Online]. Available: https://wiki.php.net/rfc/first_class_callable_syntax
- [11] N. Popov, “PHP RFC: Union Types 2.0.” Accessed: Apr. 20, 2025. [Online]. Available: https://wiki.php.net/rfc/union_types_v2

- [12] G. P. Banyard, “PHP RFC: Pure intersection types.” Accessed: Apr. 20, 2025. [Online]. Available: <https://wiki.php.net/rfc/pure-intersection-types>
- [13] G. P. Banyard and L. Garfield, “PHP RFC: Disjunctive Normal Form Types.” Accessed: Apr. 20, 2025. [Online]. Available: https://wiki.php.net/rfc/dnf_types
- [14] “Convert iterable into an internal alias for Traversable|array by Girgias · Pull Request #7309 · php/php-src.” Accessed: Apr. 20, 2025. [Online]. Available: <https://github.com/php/php-src/pull/7309>

- [15] “Subtyping.” Nov. 30, 2024. Accessed: Apr. 20, 2025. [Online]. Available: <https://en.wikipedia.org/w/index.php?title=Subtyping&oldid=1260413110>
- [16] B. H. Liskov and J. M. Wing, “A behavioral notion of subtyping,” *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 6, pp. 1811–1841, Nov. 1994, doi: 10.1145/197320.197383.
- [17] M. Seemann, “The Liskov Substitution Principle as a profunctor.” Accessed: Oct. 10, 2022. [Online]. Available: <https://blog.ploeh.dk/2021/12/06/the-liskov-substitution-principle-as-a-profunctor/>
- [18] B. Stroustrup, “Use of Protected Members,” *The C++ programming language*. Addison-Wesley, Reading, Mass, p. 405, 1997.

- [19] E. Baniassad and A. J. Summers, “Reframing the Liskov substitution principle through the lens of testing,” in *Proceedings of the 2021 ACM SIGPLAN International Symposium on SPLASH-E*, in SPLASH-E 2021. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 49–58. doi: 10.1145/3484272.3484965.
- [20] A. Le Blanc, D. Rethans, and L. Garfield, “State of Generics and Collections.” Accessed: Apr. 20, 2025. [Online]. Available: <https://thephp.foundation/blog/2024/08/19/state-of-generics-and-collections/>