PHP's Type System Dissected

Understanding how PHP's type system works

Gina P. Banyard

The PHP Foundation

2025-09-19

About me

- 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`

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.

— Wikipidia [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]

PHP's Type System

- 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 singleton 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.

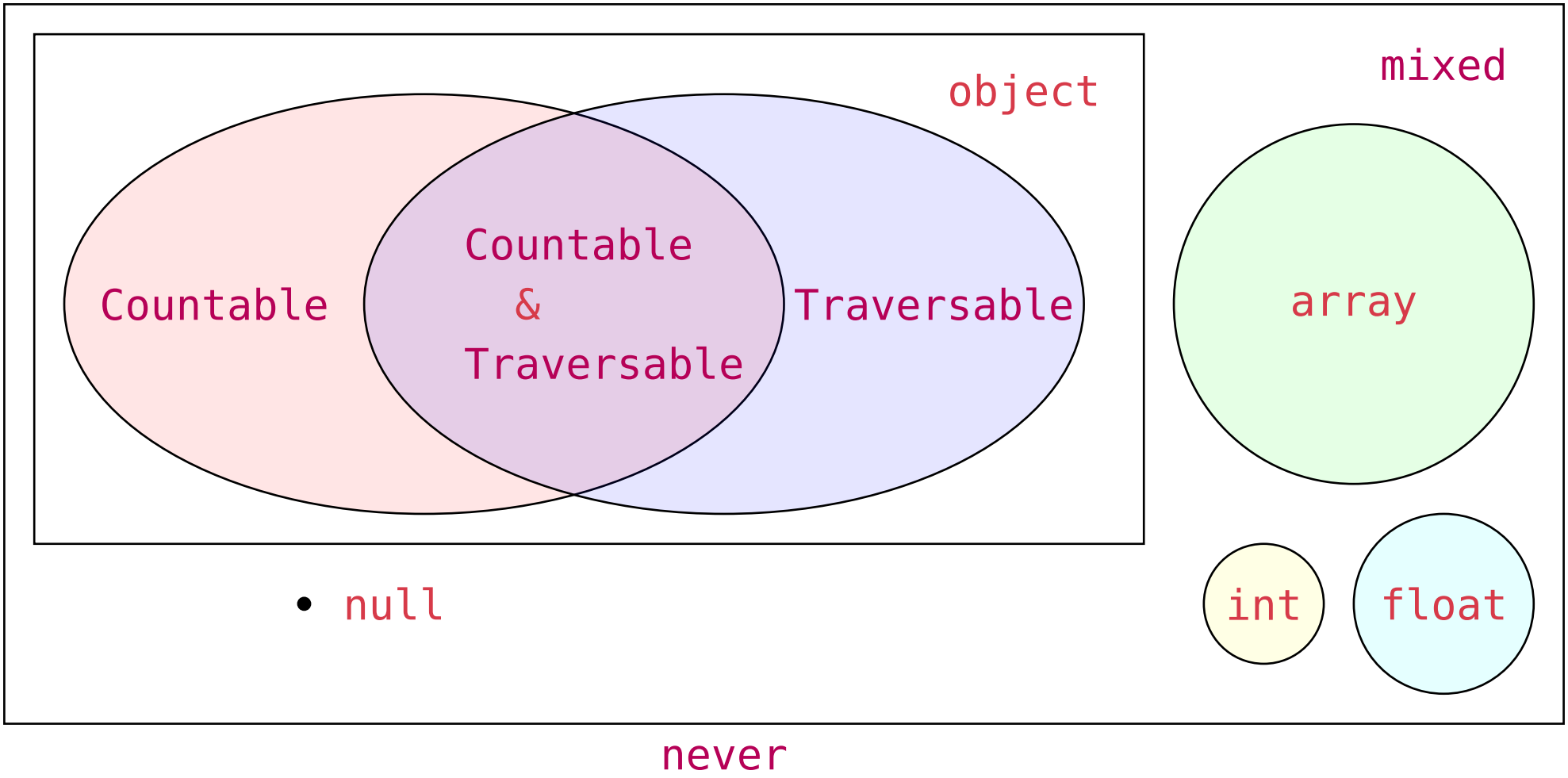
Simple union type (PHP 8.0[11]) `T|S`

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

DNF[1] types (PHP 8.2[13]) `(A&B)|(X&Y)`

[1] In 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



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

Type aliases

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

$$\texttt{iterable} := \texttt{Traversable}|\texttt{array}$$

Before it was a *pseudo*-type

Warning

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

Internal representation: Values are `zval`s

```c
struct _zval_struct {
    zend_value value;
    uint32_t type_info;
};
```

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

Internal representation: `zend_type`

`ptr` is either a class name

```c
typedef struct {
  void *ptr;
  /* Bit-mask of primitive
     types and type info */
  uint32_t type_mask;
} zend_type;
```

Internal representation: `zend_type`

`ptr` is either a class name

```c
typedef struct {
  void *ptr;
  /* Bit-mask of primitive
     types and type info */
  uint32_t type_mask;
} zend_type;
```

or a list of type

```c
typedef struct {
  uint32_t num_types;
  zend_type types[1];
} 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 $\boldsymbol{T}$.
>
> Then $\varphi(y)$ should be *true* for objects y of type $\boldsymbol{S}$ where $\boldsymbol{S}$ is a **subtype** of $\boldsymbol{T}$.
>
> — B. H. Liskov and J. M. Wing [16]

LSP: Visualized

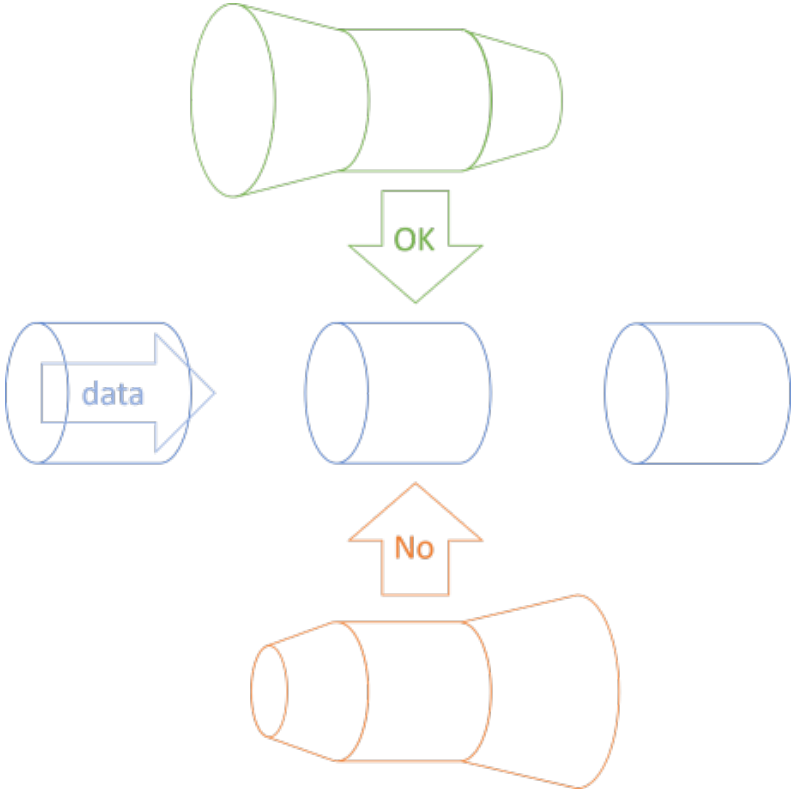


Figure 1: Visualization of Liskov Substitution Principle as a pipe. [17]

LSP: Simplified

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

An *invariant* is a predicate over **single** states $\varphi\left(x_{\psi}\right)$

A *constraint* is a predicate over **pairs** of states $\varphi\left(x_{\rho}, x_{\psi}\right)$

LSP: Effects on signatures

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 $T := T_1 | T_2 | ... | T_n$

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

if V is a union type:

$$U_1 \mid ... \mid U_n \sqsubseteq V_1 \mid ... \mid V_m \Longleftarrow \forall U_i, \exists V_j : U_i \sqsubseteq V_j$$

if V is an intersection type:

$$U_1 \mid ... \mid U_n \sqsubseteq V_1 \& ... \& V_m \Longleftarrow \forall U_i, \forall V_j : U_i \sqsubseteq V_j$$

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

if V is a union type:

$$U_1 \mid ... \mid U_n \sqsubseteq V_1 \mid ... \mid V_m \Longleftarrow \forall U_i, \exists V_j : U_i \sqsubseteq V_j$$

if V is an intersection type:

$$U_1 \mid ... \mid U_n \sqsubseteq V_1 \& ... \& V_m \Longleftarrow \forall U_i, \forall V_j : U_i \sqsubseteq V_j$$

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

Approximation in PHP


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

if V is a union type:

$$U_1 \& ... \& U_n \sqsubseteq V_1 \mid ... \mid V_m \Longleftarrow \exists V_j, \exists U_i : U_i \sqsubseteq V_j$$

if V is an intersection type:

$$U_1 \& ... \& U_n \sqsubseteq V_1 \& ... \& V_m \Longleftarrow \forall V_j, \exists U_i : U_i \sqsubseteq V_j$$

$U \sqsubseteq V$ with U an intersection type

$\forall$ means "For All", $\exists$ means "There exists".

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

if V is a union type:

$$U_1 \& ... \& U_n \sqsubseteq V_1 \mid ... \mid V_m \Longleftarrow \exists V_j, \exists U_i : U_i \sqsubseteq V_j$$

if V is an intersection type:

$$U_1 \& ... \& U_n \sqsubseteq V_1 \& ... \& V_m \Longleftarrow \forall V_j, \exists U_i : U_i \sqsubseteq V_j$$

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

if V is a union type:

$$U_1 \& ... \& U_n \sqsubseteq V_1 \mid ... \mid V_m \Longleftarrow \exists V_j, \exists U_i : U_i \sqsubseteq V_j$$

if V is an intersection type:

$$U_1 \& ... \& U_n \sqsubseteq V_1 \& ... \& V_m \Longleftarrow \forall V_j, \exists U_i : U_i \sqsubseteq V_j$$

Approximation in PHP


```
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; }
```

LSP Reframed in terms of testing

> 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

```
typedef numeric int|float
```

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

Singleton types

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

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

Function types

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 {}
```

Generics

```
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

```php
class Collection<T> {
    private array<T> $stack = [];
    public function push(T $v) {
        $this->stack[] = $v;
    }
    public function pop(): T {
      return array_pop($this->stack);
    }
}
```

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

Abstract Generics


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

In-out parameters

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)
```

In-out parameters

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

Effect types

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`

Dependent types

Dependent types are types whose definition also depend on a *value*.

Examples

$T :=$ `range<5..20>` would mean T is an integer i such that $5 \leq i \leq 20$.

$S :=$ `range<5..^20>` would mean S is an integer j such that $5 \leq j < 20$.

$U :=$ `range<5^..>` would mean U is an integer k such that $5 < k$.

A dependent string type `NonEmpty` := `string[length: range<1..>]` would be a string with at least one byte.

A dependent string type `AsciiString` := `string[bytes: range<0, 127>]` would represent a string containing only byte values between 0 and 127.

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/