

Objects

Obj, Factory, Hydrator, Serialization, JsonSerializer and Debug of Derafu Support

Anonymous · 08/10/2026

Objects

Obj

Looks at an object and fills its public properties.

Method	What it does
<code>Obj::fill(\$instance, \$data)</code>	Sets the public properties from an array. Gives the object back.
<code>Obj::getPublicProperties(\$instance)</code>	The public properties, as ReflectionProperty by name.
<code>Obj::getPublicValues(\$instance)</code>	The values of the public properties.
<code>Obj::hasProperty(\$instance, \$property, \$public = true)</code>	Whether the property exists (only public ones by default).
<code>Obj::getClassName(\$instance)</code>	The name of the class.
<code>Obj::getReflection(\$instance)</code>	A <code>ReflectionClass</code> of the object.

```
use Derafu\Support\Obj;

class Q
{
    public $x = 1;
    protected $y = 2;
    private $z = 3;
    public function __construct(public int $w = 4) {}
}

$q = new Q();

Obj::fill($q, ['x' => 10]);
Obj::getPublicValues($q); // ['x' => 10, 'w' => 4]
array_keys(Obj::getPublicProperties($q)); // ['x', 'w']
Obj::hasProperty($q, 'y'); // false: it is not public.
Obj::hasProperty($q, 'y', false); // true
Obj::getClassName($q); // "Q"
```

Factory and Hydrator

Hydrator fills an object from an array using its setters, and Factory creates the object and hydrates it.

```
use Derafu\Support\Factory;
use Derafu\Support\Hydrator;

$object = Factory::create(['a' => 1]); // A stdClass.
$object = Factory::create($data, MyClass::class); // A MyClass.
$objects = Factory::createMany($rows, MyClass::class); // A list of MyClass.
$object = Factory::createAndEnsureType($data, MyClass::class, MyInterface::class);

$object = Hydrator::hydrate($instance, $data); // An object that already exists.
$object = Hydrator::createAndHydrate(MyClass::class, $data);
```

For each attribute of the array, Hydrator tries, in this order:

1. The method `setXxx()` (Xxx is the attribute in StudlyCase: `first_name` is `setFirstName()`).
2. The method `setXxxAttribute()`.
3. The method `setAttribute($name, $value)`, if the class has it (it receives every attribute that the previous two did not take).
4. The property with that name. It is set directly, **also if it is not public**.

An attribute that none of them can take is a `LogicException` that says which attribute and which class. `Factory::createAndEnsureType()` also throws a `LogicException` if the object that it created is not of the expected type.

Because of the last step, do not hydrate an object with data that comes from a user as it is: the attributes are the names of its properties. Pick the fields first.

Serialization and JSON

```
use Derafu\Support\Serialization;
use Derafu\Support\JsonSerializer;

Serialization::isSerializable([1, 'a']); // true
Serialization::isSerializable('text'); // true
Serialization::isSerializable(new stdClass()); // true
Serialization::isSerializable(fn () => 1); // false: PHP does not serialize closures.
Serialization::isSerializable([fopen('php://memory', 'r')]); // false: a resource inside.
```

```
Serialization::isSerialized(serialize([1]));           // true
Serialization::isSerialized('abc');                   // false
Serialization::isSerialized('b:0;');                   // true: it is a serialized
`false`.
```

`isSerializable()` answers what PHP can do. A scalar and `null` can be serialized. An array or an object can be, if PHP can serialize it **and** what it has inside can be too. They are not serializable:

- **Resources**, open or closed: PHP would write them as `0`.
- **What PHP does not serialize**: closures, generators, anonymous classes, reflections, and the objects whose class forbids it or whose `__serialize()` or `__sleep()` fails (or warns).
- **An array or an object that has any of them inside**, in a property of any visibility, at any depth. For an object with `__serialize()`, what is searched is what that method gives, which is what PHP serializes.

Circular references are fine, PHP serializes them. `isSerialized()` says whether a text is the result of `serialize()` (it checks its format, it does not unserialize it).

`JsonSerializer::serialize($value, $flags = null, $depth = 512)` makes the JSON of a value that `json_encode()` would not handle well, going through arrays and objects recursively. An object becomes:

- what its `jsonSerialize()` gives, if it is a `JsonSerializable`;
- what its `toArray()` gives, if it has that method;
- a text in ISO 8601 if it is a `DateTimeInterface`;
- its text if it is a `Stringable`;
- an array of its public properties otherwise.

```
JsonSerializer::serialize([
    'date' => new DateTimeImmutable('2024-01-02 03:04:05'),
    'text' => $stringable,
    'data' => $objectWithToArray,
]);
// {"date": "2024-01-02T03:04:05-03:00", "text": "...", "data": {...}}
```

The flags by default make a readable output (pretty printed, without escaping unicode or slashes). A value that can not be encoded is a `JsonException`.

Debug

```
use Derafu\Support\Debug;

$info = Debug::inspect($var, 'myVar');
Debug::print($var, 'myVar');
```

`inspect()` gives an array with the `label`, the `type` (the class for an object), the `length` (of an array or a text), the `file` and `line` where it was called, the `caller`, the `timestamp`, the `memory_usage` and the `value` as text (`print_r()`, or JSON for `null` and booleans). The `caller` is the function or method that called it (`inAFunction()`, `Class::method()`), or `{main}` if it was called from the code that is outside of any function, as PHP says it in its traces. `print()` shows that array inside a `<pre>` (so it is for pages, not for the console), with the file, the line and the caller of the call to `print()`.

Last updated on 08/10/2026