

Typed Getters

Typed Getters

Anonymous · 20/10/2026

Typed Getters

`get()` returns `mixed`, because a container can hold anything. That is a problem when you know what a key holds: the type checker does not, and every use of the value needs a cast or a check.

The typed getters read a value and check its type. If the value has another type they throw, so what they return can be trusted, by you and by static analysis (PHPStan, Psalm).

```
$host = $options->requireString('transport.host'); // string
$port = $options->requireInt('transport.port');    // int
$ssl  = $options->getBool('transport.ssl', true);  // bool
```

`get*()` and `require*()`

Every type has two methods:

- `getX($key, $default = null)`: the value may not be there. It returns the value, or the default when the key does not exist or its value is `null`.
- `requireX($key)`: the value has to be there. It throws when the key does not exist or its value is `null`.

Situation	<code>getX()</code>	<code>requireX()</code>
The key does not exist, or its value is <code>null</code>	Returns the default (<code>null</code> if none).	Throws <code>MissingValueException</code> .
The value has another type	Throws <code>UnexpectedTypeException</code> .	Throws <code>UnexpectedTypeException</code> .
The value has the type	Returns it.	Returns it.

A value with the wrong type is never replaced by the default. If `port` is the string `"abc"`, `getInt('port', 993)` throws instead of answering 993: a mistake in the data must not look like a value that is not defined.

`0`, `''`, `false` and `[]` are values: `getInt('retries', 3)` returns `0` when `retries` is `0`.

The Methods

Methods	Type returned	Notes
<code>getString()</code> , <code>requireString()</code>	<code>string</code>	
<code>getInt()</code> , <code>requireInt()</code>	<code>int</code>	
<code>getFloat()</code> , <code>requireFloat()</code>	<code>float</code>	An <code>int</code> is accepted and returned as <code>float</code> .
<code>getBool()</code> , <code>requireBool()</code>	<code>bool</code>	
<code>getArray()</code> , <code>requireArray()</code>	<code>array</code>	
<code>getList()</code> , <code>requireList()</code>	<code>list</code>	An array with sequential integer keys.
<code>getEnum()</code> , <code>requireEnum()</code>	A <code>BackedEnum</code>	Accepts a case or a backing value.
<code>getObject()</code> , <code>requireObject()</code>	An object	Checks the class with <code>instanceof</code> .
<code>getContainer()</code> , <code>requireContainer()</code>	<code>static</code>	See Nested Containers .

They all take dot notation keys, like `get()`.

No Conversion

The getters do not convert: `"993"` is not an `int` and `"true"` is not a `bool`. Converting in every getter would make each one decide what is valid, and would hide the mistakes that the checks are for.

Convert once, when the data enters the container: a `Vault` can normalize a value with the `normalizer` of its schema (see [Validation](#)).

```
use Derafu\Container\Vault;

$options = new Vault(
    ['port' => '993'], // For example, from an environment variable.
    [
        'port' => [
            'types' => ['int', 'string'],
            'normalizer' => fn ($options, $value) => (int) $value,
        ],
    ],
);

$options->requireInt('port'); // 993
```

Arrays and Lists with Typed Elements

`getArray()` and `getList()` take `of:`, the type of the elements: `string`, `int`, `float`, `bool`, `array` or the name of a class or interface. Every element is checked, and the error names the offending one:

```
$bag = new Bag(['formats' => ['xml', 'json'], 'ids' => [1, 2, 'x']]);

$bag->requireList('formats', of: 'string'); // list<string>: ['xml', 'json']
$bag->getList('ids', of: 'int');
// UnexpectedTypeException: The value of ids.2 must be of type int; string given.
```

For `of: 'float'` the integers are returned as floats. For a class, the elements have to be instances of it:

```
$handlers = $bag->requireList('handlers', of: HandlerInterface::class); //
list<HandlerInterface>
```

`getList()` also rejects an array that is not a list (`['a' => 1]`), where `getArray()` accepts it.

Enums and Objects

```
enum Encryption: string
{
    case Ssl = 'ssl';
    case Tls = 'tls';
}

$bag = new Bag(['encryption' => 'ssl', 'mailer' => new SmtplibMailer()]);

$bag->requireEnum('encryption', Encryption::class); // Encryption::Ssl
$bag->requireObject('mailer', MailerInterface::class); // MailerInterface
```

`getEnum()` accepts a case of the enum or one of its backing values, and throws `UnexpectedTypeException` for any other value (`'none'` in the example).

Nested Containers

`get('transport')` returns an array for a `Bag`, and a container for the containers that wrap arrays when they are read (like `Derafu\Config\Options`). `getArray()` and `getContainer()` do not depend on that:

- `getArray('transport')` returns the array.
- `getContainer('transport')` returns a container of the same class, with the data of that key.

```
$transport = $options->requireContainer('transport');

$host = $transport->requireString('host');
$port = $transport->requireInt('port');
```

The nested container is a copy: changing it does not change the one it came from. A `Vault` does not give the nested container the part of its schema that applies to that key.

If a container has a constructor that does not take the data as its first argument, it overrides `createChild(array $data, string $key): static`, the protected method that builds the nested container.

ArrayAccess Is Not Typed

`$container['key']` always returns `mixed`: it is the contract of `ArrayAccess` in PHP. Use the typed getters when the type matters.

Static Analysis

The getters declare their types in PHPDoc, with conditional types for `of:`, so PHPStan knows what they return:

```
$bag->getString('a');           // string|null
$bag->requireInt('a');           // int
$bag->requireList('a', of: 'string'); // list<string>
$bag->getList('a', of: Foo::class);  // list<Foo>|null
$bag->requireEnum('a', Encryption::class); // Encryption
$bag->requireContainer('a');       // static
```

Exceptions

Exception	When
<code>Derafu\Container\Exception\MissingValueException</code>	A <code>require*()</code> method finds no value.
<code>Derafu\Container\Exception\UnexpectedTypeException</code>	The value, or one of its elements, has another type.

Their messages are translatable (see [Translations](#)):

```
The value of port must be of type int; string given.
The value of host is required but it is not defined.
```

A type name that is not a scalar type, `array`, a class or an interface, given as `of:`, throws

TranslatableInvalidArgumentException.

Last updated on 20/10/2026