Docs

Value Project

Composable value types, explicit scalar conversions, automatic parsing and value inspection

Anonymous · 15/11/2026 · #php

Table of Contents

Value Project 3
- ***Introduction*** 4
- ***Casting*** 6
- ***Parsing*** 10
- ***Types*** 13
- ***Inspection*** 19
- ***Exceptions and Translations*** 21

Value Project

Composable value types, explicit scalar conversions, automatic parsing and value inspection

Anonymous · 15/11/2026 · #php

Derafu Value

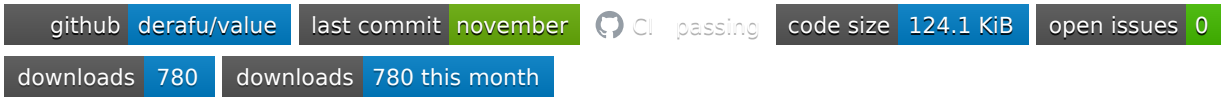
Last updated on 15/11/2026

Introduction

Validate, convert, interpret and inspect PHP values with explicit contracts

Anonymous · 15/11/2026

Value Validation and Conversion



`derafu/value` provides composable types, strict scalar reads, explicit conversions, automatic interpretation and value inspection. It requires PHP 8.5 or later and `derafu/translation`.

Installation

```
composer require derafu/value
```

Components

All four public utility classes live in `Derafu\Value`.

Class	Responsibility	Guide
`ValueCaster`	Read scalar values strictly or request explicit conversions.	Casting
`ValueParser`	Interpret individual values, with optional numeric format and precision preservation.	Parsing
`ValueType`	Compose strict checks, explicit conversions, enums, dates and collection types.	Types
`ValueInspector`	Inspect values without converting or normalizing them.	Inspection

`ValueType<T>` is a `final readonly` object. The other classes expose static methods. Errors support translation through Derafu Translation; see Exceptions and Translations.

Choose an Operation

```
use Derafu\Value\ValueCaster;
use Derafu\Value\ValueInspector;
use Derafu\Value\ValueParser;
use Derafu\Value\ValueType;
```

```
ValueCaster::requireInt(12);    // 12; rejects null and non-integers
ValueCaster::getInt(null);      // null; rejects other non-integers
ValueType::int()->parse(12);     // 12; strings would be rejected
ValueCaster::toInt('12.0');      // 12; the destination type is explicit
ValueParser::parse('12.0');      // 12; the parser chooses the interpretation
ValueInspector::isBlank('');    // true; inspection does not convert anything
```

Strict types validate the supplied PHP type. ValueCaster provides getXXX()/requireXXX() for strict scalar reads and toXXX() for explicit conversions. String reads additionally accept Stringable. The parser recognizes supported text representations and keeps unrecognized text intact. None of these operations reads array keys or assigns defaults.

Null and Collections

`ValueCaster::getXXX()` and `toXXX()` preserve null; `requireXXX()` rejects it. A conversion type such as `ValueType::toInt()` rejects it unless `nullable()` is explicitly applied. The parser returns native non-text values unchanged, including `null`, arrays, objects without `Stringable` and resources.

Collection types validate or convert their elements because that traversal is explicitly part of their type definition. `ValueParser` processes one value and does not recurse into arrays.

Last updated on 15/11/2026

Casting

Read scalar values and objects strictly or convert scalars explicitly, with nullable and required read methods

Anonymous · 15/11/2026

ValueCaster

`ValueCaster` exposes strict reads and explicit conversions. Every method accepts `mixed`; invalid input throws instead of silently becoming a default value.

Strict Reads

`getXXX()` accepts null and otherwise requires the supported input type. `requireXXX()` rejects null as well as incorrect types. Reads preserve existing types, with Stringable accepted specifically by the string methods.

Nullable read	Required read	Accepted non-null input
`getString($value): ?string`	`requireString($value): string`	string or Stringable
`getInt($value): ?int`	`requireInt($value): int`	int only
`getFloat($value): ?float`	`requireFloat($value): float`	float only
`getNumber($value): int|float|null`	`requireNumber($value): int|float`	int or float, preserving its type
`getBool($value): ?bool`	`requireBool($value): bool`	bool only
`getObject($value, $class): ?T`	`requireObject($value, $class): T`	instance of the class or interface, preserving the object

```
use Derafu\Value\Exception\TypeMismatchException;
use Derafu\Value\ValueCaster;

ValueCaster::getString(null);     // null
ValueCaster::requireString(' text '); // ' text '
ValueCaster::requireInt(12);      // 12
ValueCaster::requireNumber(12.0); // 12.0, still a float
ValueCaster::getBool(false);      // false
ValueCaster::requireObject($pool, CacheItemPoolInterface::class); // $pool

try {
```

```
    ValueCaster::getString(5);
} catch (TypeMismatchException $exception) {
    // Numeric input is not accepted by strict string reads.
}
```

String reads convert Stringable objects through the existing `toString()` operation, without trimming. Numbers and booleans are rejected by those reads. `ValueType::string()` remains a strict check of a PHP string and does not accept Stringable.

```
use Derafu\Value\ValueCaster;

$text = new class () implements Stringable {
    public function __toString(): string
    {
        return ' #ABCDEF ';
    }
};

ValueCaster::getString($text);     // ' #ABCDEF '
ValueCaster::requireString($text); // ' #ABCDEF '
ValueCaster::toString(5);          // '5': explicit conversion accepts numeric input
```

Strict read failures raise the translatable `TypeMismatchException`. A failing Stringable conversion preserves its ValueConversionException and the original throwable in the cause chain. These methods operate on one supplied value; they do not read keys or assign defaults.

Explicit Conversions

The `toXXX()` methods preserve null and convert their supported inputs. Unsupported conversions raise `ValueConversionException`.

Method	Return type
`toInt($value)`	`?int`
`toFloat($value)`	`?float`
`toNumber($value)`	`int|float|null`
`toBool($value)`	`?bool`
`toString($value)`	`?string`

Integers

`toInt()` accepts integers, integral finite floats within the platform integer range, and numeric strings representing an exact integer in that range. It rejects boolean inputs and never truncates a fraction.

```
use Derafu\Value\ValueCaster;

ValueCaster::toInt('0012');  // 12
ValueCaster::toInt('12.0');  // 12
ValueCaster::toInt('1e3');   // 1000
ValueCaster::toInt(12.0);   // 12
ValueCaster::toInt(null);   // null
```

Numeric strings are interpreted without a float intermediate. For example, `'1.0000000000000000001'` is rejected as fractional even though converting it to float could hide its fraction. Accepted integers must fit between `PHP_INT_MIN` and `PHP_INT_MAX`.

Floats

`toFloat()` accepts integers, floats and complete numeric strings according to PHP's `is_numeric()` rules. Numeric text may contain signs, decimal points, exponent notation and exterior numeric whitespace. Text such as `'12abc'`, `'1,25'`, `'1_000'`, `'INF'` and `'NAN'` is rejected. Booleans are also rejected.

```
use Derafu\Value\ValueCaster;

ValueCaster::toFloat('0012');   // 12.0
ValueCaster::toFloat('1e3');    // 1000.0
ValueCaster::toFloat('1e400');  // INF
ValueCaster::toFloat('1e-400'); // 0.0
ValueCaster::toFloat(INF);     // INF
is_nan(ValueCaster::toFloat(NAN)); // true
```

The conversion uses PHP's float representation, including rounding, infinity and underflow to zero. It does not preserve the spelling or all decimal digits of the source text. The optional policies in ValueParser can retain numeric text when interpretation would alter its format or significant decimal precision.

Numbers

`toNumber()` calls `toInt()` first. If that conversion fails, it calls `toFloat()`. Failure of the second conversion is propagated as `ValueConversionException`.

```
use Derafu\Value\ValueCaster;

ValueCaster::toNumber('12.0'); // 12
ValueCaster::toNumber(12.0);   // 12
ValueCaster::toNumber('12.5'); // 12.5
ValueCaster::toNumber(null);   // null
```

Use this method when either numeric PHP type is acceptable. It inherits both conversion contracts, including float overflow and underflow. Request `toFloat()` when the result must specifically be a float.

Booleans

Accepted input	Result
`true`, `1`, `1.0`, `'1'`, `'true'`, `'yes'`, `'on'`	`true`
`false`, `0`, `0.0`, `'0'`, `'false'`, `'no'`, `'off'`, `''`	`false`
`null`	`null`

String tokens use PHP's boolean validation filter, which trims exterior whitespace and compares case-insensitively. Whitespace-only text therefore becomes `false`. Other values, such as `2`, `0.5` or `'unknown'`, throw. This is not PHP's direct boolean cast: the text `'false'` becomes `false`.

Strings

`toString()` accepts strings, integers, floats, booleans and `Stringable` objects. Existing text is preserved without trimming. Booleans become `'1'` and `'0'`; numbers use PHP's string representation, with native `NAN` represented as `'NAN'`.

```
use Derafu\Value\ValueCaster;

ValueCaster::toString(false);     // '0'
ValueCaster::toString(true);      // '1'
ValueCaster::toString(12.5);      // '12.5'
ValueCaster::toString(' text ');  // ' text '
ValueCaster::toString(null);      // null
```

Arrays, resources and objects without `Stringable` are rejected. Numeric and boolean conversions do not accept `Stringable` objects; use the parser when their text should be interpreted automatically. A failing `__toString()` is wrapped in `ValueConversionException`, retaining the original throwable as its cause.

Last updated on 15/11/2026

Parsing

Interpret scalar text and preserve numeric format or significant decimal precision when requested

Anonymous · 15/11/2026

ValueParser

```
ValueParser::parse(
    mixed $value,
    bool $preserveNumericFormat = false,
    bool $preserveNumericPrecision = false,
): mixed
```

The parser interprets one value. `Stringable` objects are converted to text through `ValueCaster::toString()`. Strings are interpreted using the sequence below. Every other value is returned unchanged, including native integers, floats, booleans, nulls, arrays, resources and other objects.

Interpretation Order

1. Recognize the special literal representations.
2. Try `ValueCaster::toNumber()`, which tries int before float.
3. If numeric conversion fails, try `ValueCaster::toBool()`.
4. If boolean conversion also fails, return the original text.

The parser delegates conversion rules to ValueCaster. It does not define a separate numeric grammar. An unsuccessful interpretation of plain text is not an exception; a failing Stringable conversion is propagated as `ValueConversionException`.

Literal text	Result
`'null'`, `'(null)'`	`null`
`'empty'`, `'(empty)'`	`''`
`'true'`, `'(true)'`	`true`
`'false'`, `'(false)'`	`false`

These tokens are recognized case-insensitively after trimming exterior whitespace. The word `'nil'` is not recognized.

```
use Derafu\Value\ValueParser;
```

```
ValueParser::parse(' (NULL) '); // null
ValueParser::parse('empty');    // ''
ValueParser::parse('0012');     // 12
ValueParser::parse('12.0');     // 12
ValueParser::parse('12.5');     // 12.5
ValueParser::parse('1e3');      // 1000
ValueParser::parse('yes');      // true
ValueParser::parse('0');        // 0, not false
ValueParser::parse('1');        // 1, not true
ValueParser::parse('   ');      // false, through toBool()
ValueParser::parse('');         // false, unlike the literal 'empty'
ValueParser::parse(' text ');   // ' text '
```

Numeric Format Preservation

With `preserveNumericFormat: true`, a successful numeric conversion is returned only if converting the number back through `ValueCaster::toString()` reproduces the complete original text. Otherwise the original text is retained. Exterior whitespace is part of this comparison.

```
use Derafu\Value\ValueParser;

ValueParser::parse('0012', preserveNumericFormat: true);  // '0012'
ValueParser::parse('19.90', preserveNumericFormat: true); // '19.90'
ValueParser::parse('12.0', preserveNumericFormat: true);  // '12.0'
ValueParser::parse('1e3', preserveNumericFormat: true);   // '1e3'
ValueParser::parse(' 12 ', preserveNumericFormat: true);  // ' 12 '
ValueParser::parse('12', preserveNumericFormat: true);    // 12
ValueParser::parse('0.5', preserveNumericFormat: true);   // 0.5
```

This comparison follows ValueCaster's PHP number-to-string representation. It is a numeric policy: boolean words and special literals are still interpreted normally. Preserved numeric text does not fall through to boolean conversion.

Numeric Precision Preservation

With `preserveNumericPrecision: true`, non-finite numeric results and floats that fail the significant-decimal-digit comparison retain their original text. Successfully converted integers are exact and need no float comparison.

For floats, the parser compares significant mantissa digits with the shortest decimal representation found by its formatting round trip. Signs, the decimal point and insignificant leading/trailing zeros are excluded from the digit comparison. This is a decimal interpretation policy, not a requirement that fractions be exactly representable in binary.

```
use Derafu\Value\ValueParser;
```

```
ValueParser::parse('1.0000000000000000001', preserveNumericPrecision: true);
// '1.0000000000000000001'

ValueParser::parse('1e400', preserveNumericPrecision: true);   // '1e400'
ValueParser::parse('1e-400', preserveNumericPrecision: true);  // '1e-400'
ValueParser::parse('0.1', preserveNumericPrecision: true);     // 0.1
ValueParser::parse('12.5000', preserveNumericPrecision: true); // 12.5
ValueParser::parse('0012', preserveNumericPrecision: true);    // 12
```

Without this option, conversion follows ValueCaster, including float rounding, infinity and underflow. Precision preservation alone does not preserve spelling. Both options can be enabled together:

```
use Derafu\Value\ValueParser;

ValueParser::parse(
    '0012',
    preserveNumericFormat: true,
    preserveNumericPrecision: true,
); // '0012'
```

Existing Values and Stringable Objects

```
use Derafu\Value\ValueParser;

$text = new class () implements Stringable {
    public function __toString(): string
    {
        return '0012';
    }
};

ValueParser::parse($text); // 12
ValueParser::parse($text, preserveNumericFormat: true); // '0012'
ValueParser::parse(12.0);  // 12.0; an existing float is returned unchanged
ValueParser::parse(['null', '2']); // ['null', '2']; arrays are not traversed
```

The numeric options apply to interpreted text, including Stringable text. They do not transform existing native numbers. Recursive array processing belongs to a caller that explicitly traverses the array.

Last updated on 15/11/2026

Types

Compose strict validation, scalar conversion, collections, enums, dates and custom types

Anonymous · 15/11/2026

ValueType

`ValueType<T>` validates or explicitly converts a value and returns it as `T`. PHPStan follows the generic return type through collection and union compositions.

```
use Derafu\Value\ValueType;

ValueType::int()->parse(12);       // 12
ValueType::toInt()->parse('12.0'); // 12
ValueType::number()->parse(12.0);  // 12.0, unchanged
ValueType::toNumber()->parse(12.0); // 12
```

Available Factories

Factory	Contract
`mixed()`	Any value, including null.
`string()`, `int()`, `float()`, `bool()`	Exactly the indicated PHP type, without coercion.
`number()`	An existing integer or float, preserved unchanged.
`scalar()`	An existing boolean, integer, float or string.
`nonEmptyString()`	A string whose content is nonempty before and after PHP `trim()`.
`toInt()`, `toFloat()`, `toNumber()`, `toBool()`, `toString()`	Delegate to the corresponding ValueCaster method, excluding null.
`array()`	Any PHP array.
`map()`	An array with string keys only.
`indexed()`	An array with integer keys, including gaps, negative keys and lists.
`list()`	An array with consecutive integer keys starting at zero.
`arrayOf($type)`	Any array whose values match the supplied type.
`mapOf($type)`	A string-keyed array whose values match the supplied type.

Factory	Contract
`indexedOf($type)`	An integer-keyed array whose values are parsed by the supplied type.
`listOf($type)`	A list whose values match the supplied type.
`literal($first, ...$others)`	One of the supplied boolean, integer or string values, compared strictly.
`instanceOf($class)`	An object of the given class or interface, returned unchanged.
`classString($base = null)`	An existing class or enum name, optionally constrained to a base class or interface.
`backedEnum($class)`	An existing case or an exact int/string backing value of the requested backed enum.
`dateTime()`	A DateTimeInterface, integer timestamp or non-blank date string interpreted by PHP.
`oneOf($first, ...$others)`	The first type that accepts the value.

The primitive `float()` rejects integers even though `ValueCaster::toFloat()` can convert them. Use an explicit conversion type when conversion is intended.

Nullability

Apply `nullable()` to accept null in addition to a type's ordinary values:

```
use Derafu\Value\ValueType;

ValueType::int()->nullable()->parse(null);    // null
ValueType::toInt()->nullable()->parse(null);  // null
```

ValueCaster preserves null directly, but conversion types reject it by default. `ValueType::toInt()->parse(null)` throws `TypeMismatchException`. This keeps nullability explicit in a composed type.

Collections

Empty collections are accepted. Lists require consecutive integer keys starting at zero; indexed arrays require integer keys without a continuity requirement; maps require string keys; arbitrary arrays accept both. PHP converts some numeric string keys to integers, so `['123' => 'value']` is not a map, while `['01' => 'value']` is.

```
use Derafu\Value\ValueType;
```

```
$type = ValueType::arrayOf(ValueType::mapOf(ValueType::toInt()));
$type->parse([3 => ['quantity' => '12.0'], 8 => ['quantity' => '2']]);
// [3 => ['quantity' => 12], 8 => ['quantity' => 2]]
```

Collections preserve their keys and report nested errors with paths such as `rows.8.quantity`. These factories describe element types; `mapOf()` does not define required or optional named fields.

Indexed Arrays

`indexed()` validates `array<int, mixed>`. `indexedOf($type)` validates the same integer keys and parses each value with the supplied type, returning `array<int, T>`. Keys may start anywhere, have gaps, or be negative. Their order is preserved; rows are not reindexed.

```
$rows = ValueType::indexedOf(ValueType::array())->parse([
    8 => ['name' => 'First'],
    2 => ['name' => 'Last'],
]);
// [8 => ['name' => 'First'], 2 => ['name' => 'Last']]

ValueType::indexed()->parse([]); // []
ValueType::indexed()->parse([0 => 'a', 1 => 'b']); // a list is also indexed
ValueType::indexed()->parse(['extra' => 'value']); // TypeMismatchException

ValueType::indexedOf(ValueType::toInt())->parse([8 => '12.0', 2 => '1e3']);
// [8 => 12, 2 => 1000]
```

String keys are rejected without conversion. PHP itself converts a key such as `'123'` to the integer `123` when creating an array, while `'02'` remains a string and is rejected. `indexed()` preserves values unchanged; `indexedOf()` follows the supplied value type's validation or conversion policy. `nullable()` permits `null` for the collection or, when applied to the item type, for its values.

A string key produces a translatable type error with expected type `int key`, the key's path (for example `rows.extra`), and the supplied source. Value errors retain the row key in their path, such as `rows.8.quantity`.

Unions and Literal Values

```
use Derafu\Value\ValueType;

$type = ValueType::oneOf(ValueType::toInt(), ValueType::string());
$type->parse('12.0'); // 12
$type->parse('text'); // 'text'

ValueType::literal('asc', 'desc')->parse('asc'); // 'asc'
ValueType::literal(false)->parse(false);        // false; 0 would be rejected
```

Union order matters because conversion types can interpret the input. `oneOf()` tries the next candidate when the current candidate throws a translatable invalid-argument exception. It does not catch arbitrary runtime failures.

Class Names

`classString()` validates a class name without constructing an object. Input must be a string naming an existing class or enum; lookup permits autoloading. Abstract classes are accepted. Interfaces and traits are not accepted as input values.

An optional base class or interface constrains which classes are accepted:

```
use Derafu\Value\ValueType;

ValueType::classString()->parse(stdClass::class); // 'stdClass'
ValueType::classString(Exception::class)->parse(RuntimeException::class);
ValueType::classString(DateTimeInterface::class)->parse(DateTimeImmutable::class);

enum Status: string
{
    case Pending = 'pending';
    case Approved = 'approved';
}

$enumClass = ValueType::classString(BackedEnum::class)->parse(Status::class);
$enumClass::cases(); // [Status::Pending, Status::Approved]
```

The result retains its generic type: `classString(BackedEnum::class)` produces a `ValueType<class-string<BackedEnum>>`. Without a base, the result is a `class-string`. Names are returned unchanged, including their case and any leading namespace separator. Padded names, objects (including `Stringable`), and values that do not satisfy the class constraint throw `TypeMismatchException` with the path and source passed to `parse()`.

Collection factories compose with this type. For example, `ValueType::mapOf(ValueType::classString(BackedEnum::class))` validates a map of enum class names. `nullable()` explicitly permits null.

`classString(BackedEnum::class)` validates the enum class itself; `backedEnum(Status::class)` resolves a particular case or backing value.

Backed Enums

```
use Derafu\Value\ValueType;
```

```
enum Status: string
{
    case Pending = 'pending';
    case Approved = 'approved';
}

$type = ValueType::backedEnum(Status::class);
$type->parse('approved');       // Status::Approved
$type->parse(Status::Pending);  // the same enum case
```

Int-backed enums accept integer backing values; string-backed enums accept strings. Numeric strings are not coerced to integers, and another enum's case is not accepted just because it has the same backing value. Unknown values throw `TypeMismatchException`. Passing a class that is not a backed enum to the factory throws `ValueException`.

Dates

`dateTime()` preserves an existing `DateTimeInterface`. An integer timestamp becomes a `DateTimeImmutable` using `@timestamp`; date text is parsed by `DateTimeImmutable`. Relative dates, timezones and date normalization follow PHP's parser. Empty or invalid input throws `TypeMismatchException`.

```
use Derafu\Value\ValueType;

$date = ValueType::dateTime()->parse('2026-01-15');
$date->format('Y-m-d'); // '2026-01-15'
```

Custom Types and Error Context

The constructor takes a diagnostic name and a closure receiving the value, path and optional source. Return the typed result or throw `ValueType::mismatch()`:

```
use Derafu\Value\ValueType;

$even = new ValueType(
    'even',
    static fn (mixed $value, string $path, ?string $source): int
        => is_int($value) && $value % 2 === 0
            ? $value
            : throw ValueType::mismatch($path, 'even', $value, $source),
);

$even->parse(4, 'workers', 'settings.php'); // 4
```

`parse($value, $path = '', $source = null)` attaches a diagnostic location and origin to failures. An empty root path is displayed as `$`. `ValueType::path($parent, $key)` joins a diagnostic path with a string or integer key; it does not navigate data. See Exceptions and Translations.

Last updated on 15/11/2026

Inspection

Inspect blank values without coercion, whitespace normalization or recursive traversal

Anonymous · 15/11/2026

ValueInspector

`ValueInspector::isBlank(mixed $value): bool` returns true for exactly three values: `null`, `''` and `[]`.

```
use Derafu\Value\ValueInspector;

ValueInspector::isBlank(null);   // true
ValueInspector::isBlank('');     // true
ValueInspector::isBlank([]);     // true
ValueInspector::isBlank(false);  // false
ValueInspector::isBlank(0);      // false
ValueInspector::isBlank('0');    // false
ValueInspector::isBlank('   ');  // false
ValueInspector::isBlank([null]); // false
```

This is an inspection of the supplied value, not PHP's `empty()` behavior. It does not determine whether a key exists, whether a value was supplied by a caller or whether an application requires it.

Whitespace is not trimmed, array elements are not inspected recursively, and objects are neither counted nor converted to text. Even an empty `Countable` object is treated as an object rather than as `[]`.

Inspection and Interpretation

The parser performs conversions; the inspector does not:

```
use Derafu\Value\ValueInspector;
use Derafu\Value\ValueParser;

ValueInspector::isBlank('null'); // false; text exists
ValueParser::parse('null');      // null; a recognized literal
ValueInspector::isBlank(' ');    // false; the string contains a space
ValueParser::parse(' ');         // false; boolean interpretation
```

Choose the operation explicitly. A blank check does not imply a type conversion, and parsing does not decide application-level requiredness.

Last updated on 15/11/2026

Exceptions and Translations

Catch typed value failures, preserve their causes and translate package exceptions

Anonymous · 15/11/2026

Exceptions and Translations

All package exceptions are translatable and can be caught as PHP's `InvalidArgumentException`.

Exception in Derafu\Value\Exception	Parent	Cause
`ValueException`	`TranslatableInvalidArgumentException`	Invalid value-type definitions, such as a non-backed enum class.
`TypeMismatchException`	`ValueException`	A value does not match a ValueType definition or a strict ValueCaster read.
`ValueConversionException`	`TranslatableInvalidArgumentException`	A requested scalar conversion fails.

`ValueConversionException` is separate from `ValueException`. Catch the specific classes or their common `TranslatableInvalidArgumentException` parent when handling both categories.

Strict `ValueCaster::getXXX()` / `requireXXX()` methods reuse ValueType validation and raise TypeMismatchException for rejected input. If a Stringable read fails during conversion, the cause chain contains ValueConversionException followed by the original throwable. `ValueType::string()` itself still accepts PHP strings only.

Conversion Failures

```
use Derafu\Value\Exception\ValueConversionException;
use Derafu\Value\ValueCaster;

try {
    ValueCaster::toInt('text');
} catch (ValueConversionException $exception) {
    echo $exception->getMessage();
    // A value of type string cannot be converted to int.
}
```

`toNumber()` propagates the exception from its final `toFloat()` attempt if both conversions fail. A failing Stringable conversion is wrapped with the original throwable available through `getPrevious()`.

Type Failures and Causes

```
use Derafu\Value\Exception\TypeMismatchException;
use Derafu\Value\ValueType;

try {
    ValueType::toInt()->parse('text', 'quantity', 'settings.php');
} catch (TypeMismatchException $exception) {
    echo $exception->getMessage();
    // The value at "quantity" of settings.php must be of type int, string given.

    $cause = $exception->getPrevious(); // ValueConversionException
}
```

Conversion types wrap ValueConversionException as TypeMismatchException to add location and source context, retaining the original cause. Strict type mismatches do not need an underlying conversion failure. A missing path is displayed as `$`; a source is included only when one is supplied.

Spanish Translations

The package supplies `resources/translations/errors+intl-icu.es.php` through `Derafu\Value\Translation\ValueTranslationResourceProvider`. The provider supplies resources; an application creates and configures its translator.

```
use Derafu\Translation\TranslatorFactory;
use Derafu\Value\Exception\ValueConversionException;
use Derafu\Value\Translation\ValueTranslationResourceProvider;
use Derafu\Value\ValueCaster;

$translator = TranslatorFactory::create(
    defaultLocale: 'es',
    fallbackLocales: ['es', 'en'],
    resourceProviders: [new ValueTranslationResourceProvider()],
);

try {
    ValueCaster::toInt('text');
} catch (ValueConversionException $exception) {
    echo $exception->trans($translator);
    // Un valor de tipo string no se puede convertir a int.
}
```

Dependency Injection

Applications using Symfony DependencyInjection can import the optional recipe:

```
imports:
    - { resource: '../vendor/derafu/value/resources/config/value-services.yaml' }
```

It tags `ValueTranslationResourceProvider` with `derafu_translation.resource_provider`. Pass providers with that tag to `TranslatorFactory::create()` through `$resourceProviders`, following Derafu Translation. Symfony DependencyInjection is not a runtime requirement of the value utilities.

Last updated on 15/11/2026