



- [Downloads](#)
- [Documentation](#)
- [Get Involved](#)
- [Help](#)
- 

[Dutch PHP Conference 2024](#)

[Getting Started](#)

[Introduction](#)

[A simple tutorial](#)

[Language Reference](#)

[Basic syntax](#)

[Types](#)

[Variables](#)

[Constants](#)

[Expressions](#)

[Operators](#)

[Control Structures](#)

[Functions](#)

[Classes and Objects](#)

[Namespaces](#)

[Enumerations](#)

[Errors](#)

[Exceptions](#)

[Fibers](#)

[Generators](#)

[Attributes](#)

[References Explained](#)

[Predefined Variables](#)

[Predefined Exceptions](#)

[Predefined Interfaces and Classes](#)

[Predefined Attributes](#)

[Context options and parameters](#)

[Supported Protocols and Wrappers](#)

[Security](#)

[Introduction](#)

[General considerations](#)

[Installed as CGI binary](#)

[Installed as an Apache module](#)

[Session Security](#)

[Filesystem Security](#)

[Database Security](#)

[Error Reporting](#)

[User Submitted Data](#)

[Hiding PHP](#)

[Keeping Current](#)

[Features](#)

[HTTP authentication with PHP](#)

[Cookies](#)

[Sessions](#)

[Dealing with XForms](#)

[Handling file uploads](#)

[Using remote files](#)

[Connection handling](#)

[Persistent Database Connections](#)

[Command line usage](#)

[Garbage Collection](#)
[DTrace Dynamic Tracing](#)

[Function Reference](#)

[Affecting PHP's Behaviour](#)
[Audio Formats Manipulation](#)
[Authentication Services](#)
[Command Line Specific Extensions](#)
[Compression and Archive Extensions](#)
[Cryptography Extensions](#)
[Database Extensions](#)
[Date and Time Related Extensions](#)
[File System Related Extensions](#)
[Human Language and Character Encoding Support](#)
[Image Processing and Generation](#)
[Mail Related Extensions](#)
[Mathematical Extensions](#)
[Non-Text MIME Output](#)
[Process Control Extensions](#)
[Other Basic Extensions](#)
[Other Services](#)
[Search Engine Extensions](#)
[Server Specific Extensions](#)
[Session Extensions](#)
[Text Processing](#)
[Variable and Type Related Extensions](#)
[Web Services](#)
[Windows Only Extensions](#)
[XML Manipulation](#)
[GUI Extensions](#)

Keyboard Shortcuts

?	This help
j	Next menu item
k	Previous menu item
g p	Previous man page
g n	Next man page
G	Scroll to bottom
g g	Scroll to top
g h	Goto homepage
g s	Goto search (current page)
/	Focus search box

[Синтаксис атрибутов »](#)

[« Атрибуты](#)

- [Руководство по PHP](#)
- [Справочник языка](#)
- [Атрибуты](#)

Change language: Russian ▾

Введение в атрибуты

(PHP 8)

Атрибуты — это структурированные машиночитаемые метаданные, объявленные в коде. Целью атрибутов могут быть: классы (включая анонимные), методы, функции, параметры, свойства и константы класса. Затем описанные атрибутами метаданные можно проанализировать во время исполнения средствами [Reflection API](#). Поэтому атрибуты можно рассматривать как встроенный в код язык конфигурации.

Атрибуты разделяют общее и специфическое поведение сущностей в приложении. В каком-то смысле это похоже на интерфейс с его реализациями. Но интерфейсы и реализации — это про код, а атрибуты — про добавление дополнительной информации и конфигурацию. Интерфейсы могут реализовываться только классами, тогда как атрибуты можно нацеливать на методы, функции, параметры, свойства и константы классов. Поэтому атрибуты — существенно более гибкий механизм, чем интерфейсы.

Простой пример замены интерфейса с необязательными методами на код с атрибутами. Предположим, интерфейс `ActionHandler` описывает в приложении операцию, для выполнения которой одним реализациям нужна предварительная настройка, а другим — нет. И вместо внесения в интерфейс `ActionHandler` дополнительного метода `setUp()`, который для части реализаций будет пустым, можно использовать атрибут. Одним из преимуществ этого подхода является то, что мы можем использовать атрибут несколько раз.

Пример #1 Реализация опциональных методов интерфейса с помощью атрибутов

```
<?php
interface ActionHandler
{
    public function execute();
}

#[Attribute]
class SetUp {}

class CopyFile implements ActionHandler
{
    public string $fileName;
    public string $targetDirectory;

    #[SetUp]
    public function fileExists()
    {
        if (!file_exists($this->fileName)) {
            throw new RuntimeException("File does not exist");
        }
    }

    #[SetUp]
    public function targetDirectoryExists()
    {
        if (!file_exists($this->targetDirectory)) {
            mkdir($this->targetDirectory);
        } elseif (!is_dir($this->targetDirectory)) {
            throw new RuntimeException("Target directory $this->targetDirectory is not a directory");
        }
    }

    public function execute()
    {
        copy($this->fileName, $this->targetDirectory . '/' . basename($this->fileName));
    }
}
```

```
function executeAction(ActionHandler $actionHandler)
{
    $reflection = new ReflectionObject($actionHandler);

    foreach ($reflection->getMethods() as $method) {
        $attributes = $method->getAttributes(Setup::class);

        if (count($attributes) > 0) {
            $methodName = $method->getName();

            $actionHandler->$methodName();
        }
    }

    $actionHandler->execute();
}

$copyAction = new CopyFile();
$copyAction->fileName = "/tmp/foo.jpg";
$copyAction->targetDirectory = "/home/user";

executeAction($copyAction);
```

[+add a note](#)

User Contributed Notes 3 notes

[up](#)

[down](#)

25

[***Florian Krmer***](#)

1 year ago

I've tried Harshdeeps example and it didn't run out of the box and I think it is not complete, so I wrote a complete and working naive example regarding attribute based serialization.

```
<?php
declare(strict_types=1);

#[Attribute(Attribute::TARGET_CLASS_CONSTANT|Attribute::TARGET_PROPERTY)]
class JsonSerialize
{
    public function __construct(public ?string $fieldName = null) {}
}

class VersionedObject
{
    #[JsonSerialize]
    public const version = '0.0.1';
}

class UserLandClass extends VersionedObject
{
    protected string $notSerialized = 'nope';

    #[JsonSerialize('foobar')]
    public string $myValue = '';

    #[JsonSerialize('companyName')]
    public string $company = '';

    #[JsonSerialize('userLandClass')]
    protected ?UserLandClass $test;
```

```

public function __construct(?UserLandClass $userLandClass = null)
{
    $this->test = $userLandClass;
}
}

class AttributeBasedJsonSerializer {

protected const ATTRIBUTE_NAME = 'JsonSerialize';

public function serialize($object)
{
    $data = $this->extract($object);

    return json_encode($data, JSON_THROW_ON_ERROR);
}

protected function reflectProperties(array $data, ReflectionClass $reflectionClass, object $object)
{
    $reflectionProperties = $reflectionClass->getProperties();
    foreach ($reflectionProperties as $reflectionProperty) {
        $attributes = $reflectionProperty->getAttributes(static::ATTRIBUTE_NAME);
        foreach ($attributes as $attribute) {
            $instance = $attribute->newInstance();
            $name = $instance->fieldName ?? $reflectionProperty->getName();
            $value = $reflectionProperty->getValue($object);
            if (is_object($value)) {
                $value = $this->extract($value);
            }
            $data[$name] = $value;
        }
    }

    return $data;
}

protected function reflectConstants(array $data, ReflectionClass $reflectionClass)
{
    $reflectionConstants = $reflectionClass->getReflectionConstants();
    foreach ($reflectionConstants as $reflectionConstant) {
        $attributes = $reflectionConstant->getAttributes(static::ATTRIBUTE_NAME);
        foreach ($attributes as $attribute) {
            $instance = $attribute->newInstance();
            $name = $instance->fieldName ?? $reflectionConstant->getName();
            $value = $reflectionConstant->getValue();
            if (is_object($value)) {
                $value = $this->extract($value);
            }
            $data[$name] = $value;
        }
    }

    return $data;
}

protected function extract(object $object)
{
    $data = [];
    $reflectionClass = new ReflectionClass($object);
    $data = $this->reflectProperties($data, $reflectionClass, $object);
    $data = $this->reflectConstants($data, $reflectionClass);
}

```

```

return $data;
}
}

$userLandClass = new UserLandClass();
$userLandClass->company = 'some company name';
$userLandClass->myValue = 'my value';

$userLandClass2 = new UserLandClass($userLandClass);
$userLandClass2->company = 'second';
$userLandClass2->myValue = 'my second value';

$serializer = new AttributeBasedJsonSerializer();
$json = $serializer->serialize($userLandClass2);

```

```
var_dump(json_decode($json, true));
```

[up](#)

[down](#)

42

[Harshdeep ¶](#)

1 year ago

While the example displays us what we can accomplish with attributes, it should be kept in mind that the main idea behind attributes is to attach static metadata to code (methods, properties, etc.).

This metadata often includes concepts such as "markers" and "configuration". For example, you can write a serializer using reflection that only serializes marked properties (with optional configuration, such as field name in serialized file). This is reminiscent of serializers written for C# applications.

That said, full reflection and attributes go hand in hand. If your use case is satisfied by inheritance or interfaces, prefer that. The most common use case for attributes is when you have no prior information about the provided object/class.

```

<?php
interface JsonSerializable
{
    public function toJson() : array;
}
?>

```

versus, using attributes,

```

<?php

#[Attribute]
class JsonSerialize
{
    public function __constructor(public ?string $fieldName = null) {}
}

class VersionedObject
{
    #[JsonSerialize]
    public const version = '0.0.1';
}

public class UserLandClass extends VersionedObject
{
    #[JsonSerialize('call it Jackson')]
    public string $myValue;
}
?>

```

The example above is a little extra convoluted with the existence of the VersionedObject class as I wished to display that with attribute mark ups, you do not need to care how the base class manages its attributes (no call to parent in overridden method).

[up](#)

[down](#)

-70

[Justin ¶](#)

1 year ago

Allowing multiple functions to be tagged with the same Attribute is promoting weird design patterns. Because now the order of the tagged functions within the class becomes relevant. The order of functions within a class should remain arbitrary.

It would be better to limit function tagging to one Attribute only. This would force people to implement one function per attribute, which can then call all the other functions they would otherwise tag with these Attribute's.

[+ add a note](#)

- [Атрибуты](#)
 - [Введение в атрибуты](#)
 - [Синтаксис атрибутов](#)
 - [Чтение атрибутов с помощью Reflection API](#)
 - [Объявление классов атрибутов](#)
- [Copyright © 2001-2024 The PHP Group](#)
- [My PHP.net](#)
- [Contact](#)
- [Other PHP.net sites](#)
- [Privacy policy](#)

