Attributes

- Attributes overview
- Attribute syntax
- Reading Attributes with the Reflection API
- Declaring Attribute Classes

Attributes overview

PHP attributes provide structured, machine-readable metadata for classes, methods, functions, parameters, properties, and constants. They can be inspected at runtime via the Reflection API, enabling dynamic behavior without modifying code. Attributes provide a declarative way to annotate code with metadata.

Attributes enable the decoupling of a feature's implementation from its usage. While interfaces define structure by enforcing methods, attributes provide metadata across multiple elements, including methods, functions, properties, and constants. Unlike interfaces, which enforce method implementations, attributes annotate code without altering its structure.

Attributes can complement or replace optional interface methods by providing metadata instead of enforced structure. Consider an `ActionHandler` interface that represents an operation in an application. Some implementations may require a setup step while others do not. Instead of forcing all classes implementing `ActionHandler` to define a `setUp()` method, an attribute can indicate setup requirements. This approach increases flexibility, allowing attributes to be applied multiple times when necessary.

Implementing optional methods of an interface with Attributes

```
<?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);
```

Attribute syntax

Attribute syntax consists of several key components. An attribute declaration starts with `#[` and ends with `]`. Inside, one or more attributes can be listed, separated by commas. The attribute name can be unqualified, qualified, or fully-qualified, as described in Using Namespaces Basics. Arguments to the attribute are optional and enclosed in parentheses `()`. Arguments can only be literal values or constant expressions. Both positional and named argument syntax are supported.

Attribute names and their arguments are resolved to a class, and the arguments are passed to its constructor when an instance of the attribute is requested through the Reflection API. Therefore, it is recommended to introduce a class for each attribute.

Attribute Syntax

```
<?php
// a.php
namespace MyExample;

use Attribute;
```

```
#[Attribute]
class MyAttribute
{
    const VALUE = 'value';

    private $value;

    public function __construct($value = null)
    {
        $this->value = $value;
    }
}

// b.php

namespace Another;

use MyExample\MyAttribute;

#[MyAttribute]
#[\MyExample\MyAttribute]
#[MyAttribute(1234)]
#[MyAttribute(value: 1234)]
#[MyAttribute(MyAttribute::VALUE)]
#[MyAttribute(array("key" => "value"))]
#[MyAttribute(100 + 200)]
class Thing
{
}

#[MyAttribute(1234), MyAttribute(5678)]
class AnotherThing
{
}
```

Reading Attributes with the Reflection API

To access attributes from classes, methods, functions, parameters, properties, and class constants, use the `getAttributes` method provided by the Reflection API. This method returns an array of `ReflectionAttribute` instances. These instances can be queried for the attribute name, arguments, and can be used to instantiate an instance of the represented attribute.

Separating the reflected attribute representation from its actual instance provides more control over error handling, such as missing attribute classes, mistyped arguments, or missing values. Objects of the attribute class are instantiated only after calling `ReflectionAttribute::newInstance`, ensuring that argument validation occurs at that point.

Reading Attributes using Reflection API

```
<?php

#[Attribute]
class MyAttribute
{
    public $value;

    public function __construct($value)
    {
        $this->value = $value;
    }
}

#[MyAttribute(value: 1234)]
class Thing
{
}

function dumpAttributeData($reflection) {
    $attributes = $reflection->getAttributes();

    foreach ($attributes as $attribute) {
       var_dump($attribute->getName());
```

```
        var_dump($attribute->getArguments());
        var_dump($attribute->newInstance());
    }
}

dumpAttributeData(new ReflectionClass(Thing::class));
```

ENTITY

```
string(11) "MyAttribute"
array(1) {
  ["value"]=>
  int(1234)
}
object(MyAttribute)#3 (1) {
  ["value"]=>
  int(1234)
}
```

Instead of iterating over all attributes on the reflection instance, you can retrieve only those of a specific attribute class by passing the attribute class name as an argument.

Reading Specific Attributes using Reflection API

```
<?php
#[Attribute]
class MyAttribute
{
    public $value;

    public function __construct($value)
    {
        $this->value = $value;
    }
}

#[MyAttribute(value: 1234)]
class Thing
```

```
{
}

function dumpMyAttributeData($reflection) {
    $attributes = $reflection->getAttributes(MyAttribute::class);

    foreach ($attributes as $attribute) {
       var_dump($attribute->getName());
       var_dump($attribute->getArguments());
       var_dump($attribute->newInstance());
    }
}

dumpMyAttributeData(new ReflectionClass(Thing::class));
```

Declaring Attribute Classes

It is recommended to define a separate class for each attribute. In the simplest case, an empty class with the `#[Attribute]` declaration is sufficient. The attribute can be imported from the global namespace using a `use` statement.

Simple Attribute Class

```
<?php

namespace Example;

use Attribute;

#[Attribute]
class MyAttribute
{
}
```

To restrict the types of declarations an attribute can be applied to, pass a bitmask as the first argument to the `#[Attribute]` declaration.

Using target specification to restrict where attributes can be used

```
<?php

namespace Example;

use Attribute;

#[Attribute(Attribute::TARGET_METHOD | Attribute::TARGET_FUNCTION)]
class MyAttribute
{
}
```

Declaring `MyAttribute` on another type will now throw an exception during the call to `ReflectionAttribute::newInstance`

The following targets can be specified:

Attribute::TARGET_CLASS

Attribute::TARGET_FUNCTION

Attribute::TARGET_METHOD

Attribute::TARGET_PROPERTY

Attribute::TARGET_CLASS_CONSTANT

Attribute::TARGET_PARAMETER

Attribute::TARGET_ALL

By default, an attribute can only be used once per declaration. To allow an attribute to be repeatable, specify it in the bitmask of the `#[Attribute]` declaration using the `Attribute::IS_REPEATABLE` flag.

Using IS_REPEATABLE to allow attribute on a declaration multiple times

```php
<?php

namespace Example;

use Attribute;

#[Attribute(Attribute::TARGET_METHOD | Attribute::TARGET_FUNCTION | Attribute::IS_REPEATABLE)]
class MyAttribute
{
}
```

□□□ #2
□ □□□ □□□ 2026-10-31 17:37:57 UTC
□ □□□ □□□ 2026-10-31 17:37:59 UTC