Search

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

Трейты »
« Абстрактные классы

- Руководство по PHP
- Справочник языка
- Классы и объекты

Change language: Russian

Интерфейсы объектов

Интерфейсы объектов позволяют создавать код, который указывает, какие методы должен реализовать класс, без необходимости определять, как именно они должны быть реализованы. Интерфейсы разделяют пространство имён с классами и трейтами, поэтому они не могут называться одинаково.

Интерфейсы объявляются так же, как и обычные классы, но с использованием ключевого слова `interface` вместо `class`. Тела методов интерфейсов должны быть пустыми.

Все методы, определённые в интерфейсах, должны быть общедоступными, что следует из самой природы интерфейса.

На практике интерфейсы используются в двух взаимодополняющих случаях:

- Чтобы позволить разработчикам создавать объекты разных классов, которые могут использоваться взаимозаменяемо, поскольку они реализуют один и тот же интерфейс или интерфейсы. Типичный пример - несколько служб доступа к базе данных, несколько платёжных шлюзов или разных стратегий кеширования. Различные реализации могут быть заменены без каких-либо изменений в коде, который их использует.
- Чтобы разрешить функции или методу принимать и оперировать параметром, который соответствует интерфейсу, не заботясь о том, что ещё может делать объект или как он реализован. Эти интерфейсы часто называют `Iterable`, `Cacheable`, `Renderable` и так далее, чтобы описать их поведение.

Интерфейсы могут определять магические методы, требуя от реализующих классов реализации этих методов.

> **Замечание**:
>
> Хотя они поддерживаются, использование конструкторов в интерфейсах настоятельно не рекомендуется. Это значительно снижает гибкость объекта, реализующего интерфейс. Кроме того, к конструкторам не применяются правила наследования, что может привести к противоречивому и неожиданному поведению.

`implements`

Для реализации интерфейса используется оператор `implements`. Класс должен реализовать все методы, описанные в интерфейсе, иначе произойдёт фатальная ошибка. При желании классы могут реализовывать более одного интерфейса, разделяя каждый интерфейс запятой.

Внимание

Класс, реализующий интерфейс, может использовать для своих параметров имя, отличное от имени интерфейса. Однако, начиная с PHP 8.0, в языке поддерживаются именованные аргументы, и вызывающий код может полагаться на имя параметра в интерфейсе. По этой причине настоятельно рекомендуется, чтобы разработчики использовали те же имена параметров, что и реализуемый интерфейс.

> **Замечание**:
>
> Интерфейсы могут быть унаследованы друг от друга, так же, как и классы, с помощью оператора extends.
>
> **Замечание**:
>
> Класс, реализующий интерфейс, должен объявить все методы в интерфейсе с совместимой сигнатурой. Класс может реализовывать несколько интерфейсов, которые объявляют метод с одинаковым именем. В этом случае реализация должна следовать правилам совместимости сигнатуры для всех интерфейсов. Таким образом, можно применять ковариантность и контравариантность.

Константы

Интерфейсы могут содержать константы. Константы интерфейсов работают точно так же, как и константы классов. До PHP 8.1.0 они не могли быть переопределены классом или интерфейсом, который их наследует.

Примеры

Пример #1 Пример интерфейса

```
<?php
```

```php
// Объявим интерфейс 'Template'
interface Template
{
public function setVariable($name, $var);
public function getHtml($template);
}

// Реализация интерфейса
// Это будет работать
class WorkingTemplate implements Template
{
private $vars = [];

public function setVariable($name, $var)
{
$this->vars[$name] = $var;
}

public function getHtml($template)
{
foreach($this->vars as $name => $value) {
$template = str_replace('{' . $name . '}', $value, $template);
}

return $template;
}
}

// Это не будет работать
// Fatal error: Class BadTemplate contains 1 abstract methods
// and must therefore be declared abstract (Template::getHtml)
// (Фатальная ошибка: Класс BadTemplate содержит 1 абстрактный метод
// и поэтому должен быть объявлен абстрактным (Template::getHtml))
class BadTemplate implements Template
{
private $vars = [];

public function setVariable($name, $var)
{
$this->vars[$name] = $var;
}
}
?>
```

Пример #2 Наследование интерфейсов

```php
<?php
interface A
{
public function foo();
}

interface B extends A
{
public function baz(Baz $baz);
}

// Это сработает
class C implements B
{
public function foo()
{
```

```
}

public function baz(Baz $baz)
{
}
}

// Это не сработает и выдаст фатальную ошибку
class D implements B
{
public function foo()
{
}

public function baz(Foo $foo)
{
}
}
?>
```

Пример #3 Совместимость с несколькими интерфейсами

```
<?php
class Foo {}
class Bar extends Foo {}

interface A {
public function myfunc(Foo $arg): Foo;
}

interface B {
public function myfunc(Bar $arg): Bar;
}

class MyClass implements A, B
{
public function myfunc(Foo $arg): Bar
{
return new Bar();
}
}
?>
```

Пример #4 Множественное наследование интерфейсов

```
<?php
interface A
{
public function foo();
}

interface B
{
public function bar();
}

interface C extends A, B
{
public function baz();
}

class D implements C
{
```

```
public function foo()
{
}

public function bar()
{
}

public function baz()
{
}
}
?>
```

Пример #5 Интерфейсы с константами

```
<?php
interface A
{
const B = 'Константа интерфейса';
}

// Выведет: Константа интерфейса
echo A::B;


class B implements A
{
const B = 'Константа класса';
}

// Выведет: Константа класса
// До PHP 8.1.0 этот код не будет работать,
// потому что было нельзя переопределять константы.
echo B::B;
?>
```

Пример #6 Интерфейсы с абстрактными классами

```
<?php
interface A
{
public function foo(string $s): string;

public function bar(int $i): int;
}

// Абстрактный класс может реализовывать только часть интерфейса.
// Классы, расширяющие абстрактный класс, должны реализовать все остальные.
abstract class B implements A
{
public function foo(string $s): string
{
return $s . PHP_EOL;
}
}

class C extends B
{
public function bar(int $i): int
{
return $i * 2;
}
```

```
}
?>
```

Пример #7 Одновременное расширение и внедрение

```
<?php

class One
{
/* ... */
}

interface Usable
{
/* ... */
}

interface Updatable
{
/* ... */
}

// Порядок ключевых слов здесь важен. "extends" должно быть первым.
class Two extends One implements Usable, Updatable
{
/* ... */
}
?>
```

Интерфейс, совместно с объявлениями типов, предоставляет отличный способ проверки того, что определённый объект содержит определённый набор методов. Смотрите также оператор instanceof и объявление типов.

User Contributed Notes 4 notes

33
thanhn2001 at gmail dot com ¶
12 years ago
PHP prevents interface a contant to be overridden by a class/interface that DIRECTLY inherits it. However, further inheritance allows it. That means that interface constants are not final as mentioned in a previous comment. Is this a bug or a feature?

```
<?php

interface a
{
const b = 'Interface constant';
}

// Prints: Interface constant
echo a::b;

class b implements a
{
}

// This works!!!
class c extends b
{
const b = 'Class constant';
```

```
}

echo c::b;
?>
```

up
down
17

vcnbianchi ¶

2 years ago

Just as all interface methods are public, all interface methods are abstract as well.

up
down
7

williebegoode at att dot net ¶

9 years ago

In their book on Design Patterns, Erich Gamma and his associates (AKA: "The Gang of Four") use the term "interface" and "abstract class" interchangeably. In working with PHP and design patterns, the interface, while clearly a "contract" of what to include in an implementation is also a helpful guide for both re-use and making changes. As long as the implemented changes follow the interface (whether it is an interface or abstract class with abstract methods), large complex programs can be safely updated without having to re-code an entire program or module.

In PHP coding with object interfaces (as a keyword) and "interfaces" in the more general context of use that includes both object interfaces and abstract classes, the purpose of "loose binding" (loosely bound objects) for ease of change and re-use is a helpful way to think about both uses of the term "interface." The focus shifts from "contractual" to "loose binding" for the purpose of cooperative development and re-use.

up
down
-1

xedin dot unknown at gmail dot com ¶

2 years ago

This page says that if extending multiple interfaces with the same methods, the signature must be compatible. But this is not all there is to it: the order of `extends` matters. This is a known issue, and while it is disputable whether or not it is a bug, one should be aware of it, and code interfaces with this in mind.

https://bugs.php.net/bug.php?id=67270
https://bugs.php.net/bug.php?id=76361
https://bugs.php.net/bug.php?id=80785