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

Анонимные классы »
« Интерфейсы объектов

- Руководство по PHP
- Справочник языка
- Классы и объекты

Change language: Russian

Трейты

PHP реализует способ переиспользования кода, называемый трейтами (Traits).

Трейт — это механизм переиспользования кода в языках с поддержкой одиночного наследования, к которым относится PHP. Задача трейта — уменьшить ограничения одиночного наследования, разрешая разработчику легко переиспользовать наборы методов в нескольких независимых классах, находящихся в разных иерархиях классов. Семантика комбинации трейтов и классов определена так, чтобы снизить уровень сложности, а также избежать типичных проблем, свойственных множественному наследованию и примесям (Mixins).

Трейт очень похож на класс, но рассчитан только на группировку функциональности тонко контролируемым и согласованным образом. Нельзя создать отдельный экземляр трейта. Трейт — это дополнение к обычному наследованию и инструмент построения горизонтальной композиции поведения, то есть работы с членами класса (трейта) без требования наследования.

Пример #1 Пример использования трейта

```
<?php
trait ezcReflectionReturnInfo {
function getReturnType() { /*1*/ }
function getReturnDescription() { /*2*/ }
}

class ezcReflectionMethod extends ReflectionMethod {
use ezcReflectionReturnInfo;
/* ... */
}

class ezcReflectionFunction extends ReflectionFunction {
use ezcReflectionReturnInfo;
/* ... */
}
?>
```

Приоритет

Член, унаследованный из базового класса, переопределяется членом, введённым трейтом. Порядок приоритета следующий: члены текущего класса переопределяют методы трейта, которые, со своей стороны, переопределяют унаследованные методы.

Пример #2 Пример приоритета старшинства

Наследуемый от базового класса метод переопределяется методом, добавленным в класс MyHelloWorld из трейта SayWorld. Поведение методов трейта повторяет поведение методов класса MyHelloWorld. Порядок приоритета такой: методы текущего класса переопределяют методы трейта, которые, со своей стороны, переопределяют методы базового класса.

```
<?php
class Base {
public function sayHello() {
echo 'Hello ';
}
}

trait SayWorld {
public function sayHello() {
parent::sayHello();
echo 'World!';
}
}

class MyHelloWorld extends Base {
```

```
use SayWorld;
}

$o = new MyHelloWorld();
$o->sayHello();
?>
```

Результат выполнения приведённого примера:

```
Hello World!
```

Пример #3 Пример альтернативного порядка приоритета

```
<?php
trait HelloWorld {
public function sayHello() {
echo 'Hello World!';
}
}

class TheWorldIsNotEnough {
use HelloWorld;
public function sayHello() {
echo 'Hello Universe!';
}
}

$o = new TheWorldIsNotEnough();
$o->sayHello();
?>
```

Результат выполнения приведённого примера:

```
Hello Universe!
```

Несколько трейтов

В класс можно добавить несколько трейтов, перечислив их в директиве `use` через запятую.

Пример #4 Пример использования нескольких трейтов

```
<?php
trait Hello {
public function sayHello() {
echo 'Hello ';
}
}

trait World {
public function sayWorld() {
echo 'World';
}
}

class MyHelloWorld {
use Hello, World;
public function sayExclamationMark() {
echo '!';
}
}

$o = new MyHelloWorld();
$o->sayHello();
$o->sayWorld();
$o->sayExclamationMark();
```

```
?>
```

Результат выполнения приведённого примера:

```
Hello World!
```

Разрешение конфликтов

Если два трейта добавляют метод с одним и тем же именем, будет вызвана фатальная ошибка, если конфликт явно не разрешён.

Для разрешения конфликтов именования между трейтами, включёнными в один и тот же класс, вызывают оператор `insteadof`, чтобы точно выбрать один из конфликтующих методов.

Так как предыдущий оператор только исключает методы, оператор `as` может включить один из конфликтующих методов под другим именем. Обратите внимание, что оператор `as` не переименовывает метод, а также не влияет ни на какой другой метод.

Пример #5 Пример разрешения конфликтов

В этом примере в класс Talker включены трейты A и B. Поскольку в трейтах A и B есть конфликтующие методы, класс использует вариант метода smallTalk из трейта B, а вариант метода bigTalk — из трейта A.

Класс Aliased_Talker применяет оператор `as`, чтобы использовать реализацию метода bigTalk из класса B под дополнительным псевдонимом `talk`.

```
<?php
trait A {
public function smallTalk() {
echo 'a';
}
public function bigTalk() {
echo 'A';
}
}

trait B {
public function smallTalk() {
echo 'b';
}
public function bigTalk() {
echo 'B';
}
}

class Talker {
use A, B {
B::smallTalk insteadof A;
A::bigTalk insteadof B;
}
}

class Aliased_Talker {
use A, B {
B::smallTalk insteadof A;
A::bigTalk insteadof B;
B::bigTalk as talk;
}
}
?>
```

Изменение видимости метода

Применяя оператор `as`, можно также изменить видимость метода в классе, в который включён трейт.

Пример #6 Пример изменения видимости метода

```
<?php
trait HelloWorld {
public function sayHello() {
echo 'Hello World!';
}
}

// Изменение видимости метода sayHello
class MyClass1 {
use HelloWorld { sayHello as protected; }
}

// Создание псевдонима метода с изменённой видимостью
// видимость sayHello не изменилась
class MyClass2 {
use HelloWorld { sayHello as private myPrivateHello; }
}
?>
```

Трейты, состоящие из трейтов

Трейты можно включать и в классы, и в другие трейты. Трейт может быть полностью или частично составлен из членов, описанных в других трейтах, один или несколько из которых включены в определении трейта.

Пример #7 Пример трейтов, составленных из трейтов

```
<?php
trait Hello {
public function sayHello() {
echo 'Hello ';
}
}

trait World {
public function sayWorld() {
echo 'World!';
}
}

trait HelloWorld {
use Hello, World;
}

class MyHelloWorld {
use HelloWorld;
}

$o = new MyHelloWorld();
$o->sayHello();
$o->sayWorld();
?>
```

Результат выполнения приведённого примера:

```
Hello World!
```

Абстрактные члены трейтов

Трейты поддерживают абстрактные методы, чтобы установить требования к классу, в который будет включён трейт. Поддерживаются общедоступные, защищённые и закрытые методы. До PHP 8.0.0 поддерживались только общедоступные и защищённые абстрактные методы.

Предостережение

Начиная с PHP 8.0.0 сигнатура конкретного метода должна следовать правилам совместимости сигнатур. Ранее сигнатура метода могла несовпадать.

Пример #8 Требования трейта при помощи абстрактных методов

```
<?php
trait Hello {
public function sayHelloWorld() {
echo 'Hello'.$this->getWorld();
}
abstract public function getWorld();
}

class MyHelloWorld {
private $world;
use Hello;
public function getWorld() {
return $this->world;
}
public function setWorld($val) {
$this->world = $val;
}
}
?>
```

Статические члены трейта

В трейтах можно определять статические переменные, статические методы и статические свойства.

> **Замечание**:
>
> Начиная с PHP 8.1.0 прямой вызов статического метода или прямой доступ к статическому свойству в трейте устарел. К статическим методам и свойствам нужно обращаться только в классе, в который включён трейт.

Пример #9 Статические переменные

```
<?php
trait Counter {
public function inc() {
static $c = 0;
$c = $c + 1;
echo "$c\n";
}
}

class C1 {
use Counter;
}

class C2 {
use Counter;
}

$o = new C1(); $o->inc(); // echo 1
$p = new C2(); $p->inc(); // echo 1
?>
```

Пример #10 Статические методы

```
<?php
trait StaticExample {
```

```
public static function doSomething() {
echo 'Что-либо делаем';
}
}

class Example {
use StaticExample;
}

Example::doSomething();
?>
```

Пример #11 Статические свойства

```
<?php
trait StaticExample {
public static $static = 'foo';
}
class Example {
use StaticExample;
}
echo Example::$static;
?>
```

Свойства

Трейты также могут определять свойства.

Пример #12 Определение свойств

```
<?php
trait PropertiesTrait {
public $x = 1;
}

class PropertiesExample {
use PropertiesTrait;
}

$example = new PropertiesExample;
$example->x;
?>
```

Если трейт определяет свойство, то класс не может определить свойство с таким же именем, кроме случаев полного совпадения (та же область видимости и тип, модификатор readonly и начальное значение), иначе будет выброшена фатальная ошибка.

Пример #13 Разрешение конфликтов

```
<?php
trait PropertiesTrait {
public $same = true;
public $different1 = false;
public bool $different2;
public bool $different3;
}

class PropertiesExample {
use PropertiesTrait;
public $same = true;
public $different1 = true; // Фатальная ошибка
public string $different2; // Фатальная ошибка
readonly protected bool $different3; // Фатальная ошибка
}
```

```
?>
```

Константы

Начиная с версии PHP 8.2.0 трейты могут также определять константы.

Пример #14 Определение констант

```
<?php
trait ConstantsTrait {
public const FLAG_MUTABLE = 1;
final public const FLAG_IMMUTABLE = 5;
}

class ConstantsExample {
use ConstantsTrait;
}

$example = new ConstantsExample;
echo $example::FLAG_MUTABLE; // 1
?>
```

Если трейт определяет константу, то класс не может определить константу с таким же именем, если только они не совместимы (одинаковая область видимости, начальное значение и модификатор final), иначе выбрасывается фатальная ошибка.

Пример #15 Разрешение конфликтов

```
<?php
trait ConstantsTrait {
public const FLAG_MUTABLE = 1;
final public const FLAG_IMMUTABLE = 5;
}

class ConstantsExample {
use ConstantsTrait;
public const FLAG_IMMUTABLE = 5; // Фатальная ошибка
}
?>
```


User Contributed Notes 25 notes

646
Safak Ozpinar / safakozpinar at gmail ¶
11 years ago

Unlike inheritance; if a trait has static properties, each class using that trait has independent instances of those properties.

Example using parent class:

```
<?php
class TestClass {
public static $_bar;
}
class Foo1 extends TestClass { }
class Foo2 extends TestClass { }
Foo1::$_bar = 'Hello';
Foo2::$_bar = 'World';
echo Foo1::$_bar . ' ' . Foo2::$_bar; // Prints: World World
?>
```

Example using trait:

```
<?php
trait TestTrait {
public static $_bar;
}
class Foo1 {
use TestTrait;
}
class Foo2 {
use TestTrait;
}
Foo1::$_bar = 'Hello';
Foo2::$_bar = 'World';
echo Foo1::$_bar . ' ' . Foo2::$_bar; // Prints: Hello World
?>
```

up
down
448
greywire at gmail dot com ¶
11 years ago

The best way to understand what traits are and how to use them is to look at them for what they essentially are: language assisted copy and paste.

If you can copy and paste the code from one class to another (and we've all done this, even though we try not to because its code duplication) then you have a candidate for a trait.

up
down
244
Stefan W ¶
10 years ago

Note that the "use" operator for traits (inside a class) and the "use" operator for namespaces (outside the class) resolve names differently. "use" for namespaces always sees its arguments as absolute (starting at the global namespace):

```
<?php
namespace Foo\Bar;
use Foo\Test; // means \Foo\Test - the initial \ is optional
?>
```

On the other hand, "use" for traits respects the current namespace:

```
<?php
namespace Foo\Bar;
class SomeClass {
use Foo\Test; // means \Foo\Bar\Foo\Test
}
?>
```

Together with "use" for closures, there are now three different "use" operators. They all mean different things and behave differently.

up
down
8
JustAddingSomeAdditionalUseCase ¶
11 months ago

I have not seen this specific use case:

"Wanting to preserve action of parent class method, the trait one calling ::parent & also the child class mehod action".

```
// Child class.
use SuperTrait {
initialize as initializeOr;
}
```

```
public function initialize(array &$element) {
...
$this->initializeOr($element);
}
// Trait.
public function initialize(array &$element) {
...
parent::initialize($element);
}
// Parent class.
public function initialize(array &$element) {
...
}
```

up
down
100

t8 at AT pobox dot com ¶

11 years ago

Another difference with traits vs inheritance is that methods defined in traits can access methods and properties of the class they're used in, including private ones.

```
For example:
<?php
trait MyTrait
{
protected function accessVar()
{
return $this->var;
}

}

class TraitUser
{
use MyTrait;

private $var = 'var';

public function getVar()
{
return $this->accessVar();
}
}

$t = new TraitUser();
echo $t->getVar(); // -> 'var'

?>
```

up
down
97

chris dot rutledge at gmail dot com ¶

12 years ago

It may be worth noting here that the magic constant __CLASS__ becomes even more magical - __CLASS__ will return the name of the class in which the trait is being used.

```
for example

<?php
trait sayWhere {
public function whereAmI() {
echo __CLASS__;
```

```
}
}

class Hello {
use sayWHere;
}

class World {
use sayWHere;
}

$a = new Hello;
$a->whereAmI(); //Hello

$b = new World;
$b->whereAmI(); //World
?>
```

The magic constant __TRAIT__ will giev you the name of the trait

up
down
61

qeremy (!) gmail ¶

8 years ago

Keep in mind; "final" keyword is useless in traits when directly using them, unlike extending classes / abstract classes.

```
<?php
trait Foo {
final public function hello($s) { print "$s, hello!"; }
}
class Bar {
use Foo;
// Overwrite, no error
final public function hello($s) { print "hello, $s!"; }
}

abstract class Foo {
final public function hello($s) { print "$s, hello!"; }
}
class Bar extends Foo {
// Fatal error: Cannot override final method Foo::hello() in ..
final public function hello($s) { print "hello, $s!"; }
}
?>
```

But this way will finalize trait methods as expected;

```
<?php
trait FooTrait {
final public function hello($s) { print "$s, hello!"; }
}
abstract class Foo {
use FooTrait;
}
class Bar extends Foo {
// Fatal error: Cannot override final method Foo::hello() in ..
final public function hello($s) { print "hello, $s!"; }
}
?>
```

up
down
15

rawsrc ¶
5 years ago
About the (Safak Ozpinar / safakozpinar at gmail)'s great note, you can still have the same behavior than inheritance using trait with this approach :

```
<?php

trait TestTrait {
public static $_bar;
}

class FooBar {
use TestTrait;
}

class Foo1 extends FooBar {

}
class Foo2 extends FooBar {

}
Foo1::$_bar = 'Hello';
Foo2::$_bar = 'World';
echo Foo1::$_bar . ' ' . Foo2::$_bar; // Prints: World World
```

up
down
31
canufrank ¶
7 years ago
A number of the notes make incorrect assertions about trait behaviour because they do not extend the class.

So, while "Unlike inheritance; if a trait has static properties, each class using that trait has independent instances of those properties.

Example using parent class:

```
<?php
class TestClass {
public static $_bar;
}
class Foo1 extends TestClass { }
class Foo2 extends TestClass { }
Foo1::$_bar = 'Hello';
Foo2::$_bar = 'World';
echo Foo1::$_bar . ' ' . Foo2::$_bar; // Prints: World World
?>
```

Example using trait:

```
<?php
trait TestTrait {
public static $_bar;
}
class Foo1 {
use TestTrait;
}
class Foo2 {
use TestTrait;
}
Foo1::$_bar = 'Hello';
Foo2::$_bar = 'World';
echo Foo1::$_bar . ' ' . Foo2::$_bar; // Prints: Hello World
?>"
```

shows a correct example, simply adding

```
<?php
require_once('above');
class Foo3 extends Foo2 {
}
Foo3::$_bar = 'news';
echo Foo1::$_bar . ' ' . Foo2::$_bar . ' ' . Foo3::$_bar;

// Prints: Hello news news
```

I think the best conceptual model of an incorporated trait is an advanced insertion of text, or as someone put it "language assisted copy and paste." If Foo1 and Foo2 were defined with $_bar, you would not expect them to share the instance. Similarly, you would expect Foo3 to share with Foo2, and it does.

Viewing this way explains away a lot of the 'quirks' that are observed above with final, or subsequently declared private vars,

up
down
7

yeu_ym at yahoo dot com ¶

4 years ago

Here is an example how to work with visiblity and conflicts.

```
<?php

trait A
{
private function smallTalk()
{
echo 'a';
}

private function bigTalk()
{
echo 'A';
}
}

trait B
{
private function smallTalk()
{
echo 'b';
}

private function bigTalk()
{
echo 'B';
}
}

trait C
{
public function smallTalk()
{
echo 'c';
}

public function bigTalk()
{
echo 'C';
}
}
```

```
class Talker
{
use A, B, C {
//visibility for methods that will be involved in conflict resolution
B::smallTalk as public;
A::bigTalk as public;

//conflict resolution
B::smallTalk insteadof A, C;
A::bigTalk insteadof B, C;

//aliases with visibility change
B::bigTalk as public Btalk;
A::smallTalk as public asmalltalk;

//aliases only, methods already defined as public
C::bigTalk as Ctalk;
C::smallTalk as cmallstalk;
}

}

(new Talker)->bigTalk();//A
(new Talker)->Btalk();//B
(new Talker)->Ctalk();//C

(new Talker)->asmalltalk();//a
(new Talker)->smallTalk();//b
(new Talker)->cmallstalk();//c
```

up
down
12

qschuler at neosyne dot com ¶
9 years ago

Note that you can omit a method's inclusion by excluding it from one trait in favor of the other and doing the exact same thing in the reverse way.

```
<?php

trait A {
public function sayHello()
{
echo 'Hello from A';
}

public function sayWorld()
{
echo 'World from A';
}
}

trait B {
public function sayHello()
{
echo 'Hello from B';
}

public function sayWorld()
{
echo 'World from B';
}
```

```
}

class Talker {
use A, B {
A::sayHello insteadof B;
A::sayWorld insteadof B;
B::sayWorld insteadof A;
}
}

$talker = new Talker();
$talker->sayHello();
$talker->sayWorld();

?>
```

The method sayHello is imported, but the method sayWorld is simply excluded.

up
down
43
ryan at derokorian dot com ¶
11 years ago

Simple singleton trait.

```
<?php

trait singleton {
/**
* private construct, generally defined by using class
*/
//private function __construct() {}

public static function getInstance() {
static $_instance = NULL;
$class = __CLASS__;
return $_instance ?: $_instance = new $class;
}

public function __clone() {
trigger_error('Cloning '.__CLASS__.' is not allowed.',E_USER_ERROR);
}

public function __wakeup() {
trigger_error('Unserializing '.__CLASS__.' is not allowed.',E_USER_ERROR);
}
}

/**
* Example Usage
*/

class foo {
use singleton;

private function __construct() {
$this->name = 'foo';
}
}

class bar {
use singleton;
```

```
private function __construct() {
$this->name = 'bar';
}
}

$foo = foo::getInstance();
echo $foo->name;

$bar = bar::getInstance();
echo $bar->name;
```

up
down
16
marko at newvibrations dot net ¶
7 years ago
As already noted, static properties and methods in trait could be accessed directly using trait. Since trait is language assisted c/p, you should be aware that static property from trait will be initialized to the value trait property had in the time of class declaration.

Example:

```
<?php

trait Beer {
protected static $type = 'Light';
public static function printed(){
echo static::$type.PHP_EOL;
}
public static function setType($type){
static::$type = $type;
}
}

class Ale {
use Beer;
}

Beer::setType("Dark");

class Lager {
use Beer;
}

Beer::setType("Amber");

header("Content-type: text/plain");

Beer::printed(); // Prints: Amber
Ale::printed(); // Prints: Light
Lager::printed(); // Prints: Dark

?>
```

up
down
17
Edward ¶
11 years ago
The difference between Traits and multiple inheritance is in the inheritance part. A trait is not inherited from, but rather included or mixed-in, thus becoming part of "this class". Traits also provide a more controlled means of resolving conflicts that inevitably arise when using multiple inheritance in the few languages that support them (C++). Most modern languages are going the approach of a "traits" or "mixin" style system as opposed to multiple-inheritance, largely due to the ability to control ambiguities if a method is declared in multiple "mixed-in" classes.

Also, one can not "inherit" static member functions in multiple-inheritance.

up

down

8

balbuf ¶

8 years ago

(It's already been said, but for the sake of searching on the word "relative"...)

The "use" keyword to import a trait into a class will resolve relative to the current namespace and therefore should include a leading slash to represent a full path, whereas "use" at the namespace level is always absolute.

up

down

3

guidobelluomo at gmail dot com ¶

3 years ago

If you override a method which was defined by a trait, calling the parent method will also call the trait's override. Therefore if you need to derive from a class which has a trait, you can extend the class without losing the trait's functionality:

```
<?php

trait ExampleTrait
{
public function output()
{
parent::output();
echo "bar<br>";
}
}

class Foo
{
public function output()
{
echo "foo<br>";
}
}

class FooBar extends Foo
{
use ExampleTrait;
}

class FooBarBaz extends FooBar
{
use ExampleTrait;
public function output()
{
parent::output();
echo "baz";
}
}

(new FooBarBaz())->output();
?>

Output:
foo
bar
baz
```

up

down
14
Kristof ¶
9 years ago
don't forget you can create complex (embedded) traits as well

```
<?php
trait Name {
// ...
}
trait Address {
// ...
}
trait Telephone {
// ...
}
trait Contact {
use Name, Address, Telephone;
}
class Customer {
use Contact;
}
class Invoce {
use Contact;
}
?>
```

up
down
14
D. Marti ¶
11 years ago
Traits are useful for strategies, when you want the same data to be handled (filtered, sorted, etc) differently.

For example, you have a list of products that you want to filter out based on some criteria (brands, specs, whatever), or sorted by different means (price, label, whatever). You can create a sorting trait that contains different functions for different sorting types (numeric, string, date, etc). You can then use this trait not only in your product class (as given in the example), but also in other classes that need similar strategies (to apply a numeric sort to some data, etc).

```
<?php
trait SortStrategy {
private $sort_field = null;
private function string_asc($item1, $item2) {
return strnatcmp($item1[$this->sort_field], $item2[$this->sort_field]);
}
private function string_desc($item1, $item2) {
return strnatcmp($item2[$this->sort_field], $item1[$this->sort_field]);
}
private function num_asc($item1, $item2) {
if ($item1[$this->sort_field] == $item2[$this->sort_field]) return 0;
return ($item1[$this->sort_field] < $item2[$this->sort_field] ? -1 : 1 );
}
private function num_desc($item1, $item2) {
if ($item1[$this->sort_field] == $item2[$this->sort_field]) return 0;
return ($item1[$this->sort_field] > $item2[$this->sort_field] ? -1 : 1 );
}
private function date_asc($item1, $item2) {
$date1 = intval(str_replace('-', '', $item1[$this->sort_field]));
$date2 = intval(str_replace('-', '', $item2[$this->sort_field]));
if ($date1 == $date2) return 0;
return ($date1 < $date2 ? -1 : 1 );
}
private function date_desc($item1, $item2) {
```

```
$date1 = intval(str_replace('-', '', $item1[$this->sort_field]));
$date2 = intval(str_replace('-', '', $item2[$this->sort_field]));
if ($date1 == $date2) return 0;
return ($date1 > $date2 ? -1 : 1 );
}
}

class Product {
public $data = array();

use SortStrategy;

public function get() {
// do something to get the data, for this ex. I just included an array
$this->data = array(
101222 => array('label' => 'Awesome product', 'price' => 10.50, 'date_added' => '2012-02-01'),
101232 => array('label' => 'Not so awesome product', 'price' => 5.20, 'date_added' => '2012-03-20'),
101241 => array('label' => 'Pretty neat product', 'price' => 9.65, 'date_added' => '2012-04-15'),
101256 => array('label' => 'Freakishly cool product', 'price' => 12.55, 'date_added' => '2012-01-11'),
101219 => array('label' => 'Meh product', 'price' => 3.69, 'date_added' => '2012-06-11'),
);
}

public function sort_by($by = 'price', $type = 'asc') {
if (!preg_match('/^(asc|desc)$/', $type)) $type = 'asc';
switch ($by) {
case 'name':
$this->sort_field = 'label';
uasort($this->data, array('Product', 'string_'.$type));
break;
case 'date':
$this->sort_field = 'date_added';
uasort($this->data, array('Product', 'date_'.$type));
break;
default:
$this->sort_field = 'price';
uasort($this->data, array('Product', 'num_'.$type));
}
}
}

$product = new Product();
$product->get();
$product->sort_by('name');
echo '<pre>'.print_r($product->data, true).'</pre>';
?>
```

up
down
3

bscheshirwork at gmail dot com ¶

6 years ago

https://3v4l.org/mFuQE

```
1. no deprecate if same-class-named method get from trait
2. replace same-named method ba to aa in C

trait ATrait {
public function a(){
return 'Aa';
}
}
```

```
trait BTrait {
public function a(){
return 'Ba';
}
}

class C {
use ATrait{
a as aa;
}
use BTrait{
a as ba;
}

public function a() {
return static::aa() . static::ba();
}
}

$o = new C;
echo $o->a(), "\n";

class D {
use ATrait{
ATrait::a as aa;
}
use BTrait{
BTrait::a as ba;
}

public function a() {
return static::aa() . static::ba();
}
}

$o = new D;
echo $o->a(), "\n";

class E {
use ATrait{
ATrait::a as aa;
ATrait::a insteadof BTrait;
}
use BTrait{
BTrait::a as ba;
}

public function e() {
return static::aa() . static::ba();
}
}

$o = new E;
echo $o->e(), "\n";

class F {
use ATrait{
a as aa;
}
use BTrait{
a as ba;
}
```

```
public function f() {
return static::aa() . static::ba();
}
}

$o = new F;
echo $o->f(), "\n";

AaAa
AaBa

Deprecated: Methods with the same name as their class will not be constructors in a future version of PHP; E has a deprecated constructor in /in/mFuQE on line 48
AaBa

Fatal error: Trait method a has not been applied, because there are collisions with other trait methods on F in /in/mFuQE on line 65
```

up
down
5
cody at codysnider dot com ¶
6 years ago

```
/*
DocBlocks pertaining to the class or trait will NOT be carried over when applying the trait.

Results trying a couple variations on classes with and without DocBlocks that use a trait with a DocBlock
*/

<?php

/**
* @Entity
*/
trait Foo
{
protected $foo;
}

/**
* @HasLifecycleCallbacks
*/
class Bar
{
use \Foo;

protected $bar;
}

class MoreBar
{
use \Foo;

protected $moreBar;
}

$w = new \ReflectionClass('\Bar');
echo $w->getName() . ":\r\n";
echo $w->getDocComment() . "\r\n\r\n";

$x = new \ReflectionClass('\MoreBar');
echo $x->getName() . ":\r\n";
```

```
echo $x->getDocComment() . "\r\n\r\n";

$barObj = new \Bar();
$y = new \ReflectionClass($barObj);
echo $y->getName() . ":\r\n";
echo $y->getDocComment() . "\r\n\r\n";

foreach($y->getTraits() as $traitObj) {
echo $y->getName() . " ";
echo $traitObj->getName() . ":\r\n";
echo $traitObj->getDocComment() . "\r\n";
}

$moreBarObj = new \MoreBar();
$z = new \ReflectionClass($moreBarObj);
echo $z->getName() . " ";
echo $z->getDocComment() . "\r\n\r\n";

foreach($z->getTraits() as $traitObj) {
echo $z->getName() . " ";
echo $traitObj->getName() . ":\r\n";
echo $traitObj->getDocComment() . "\r\n";
}
```

up
down
4

Carlos Alberto Bertholdo Carucce ¶

7 years ago

If you want to resolve name conflicts and also change the visibility of a trait method, you'll need to declare both in the same line:

```
trait testTrait{

public function test(){
echo 'trait test';
}

}

class myClass{

use testTrait {
testTrait::test as private testTraitF;
}

public function test(){
echo 'class test';
echo '<br/>';
$this->testTraitF();
}

}

$obj = new myClass();
$obj->test(); //prints both 'trait test' and 'class test'
$obj->testTraitF(); //The method is not accessible (Fatal error: Call to private method myClass::testTraitF() )
```

up
down
3

katrinaelaine6 at gmail dot com ¶

6 years ago

Adding to "atorich at gmail dot com":

The behavior of the magic constant __CLASS__ when used in traits is as expected if you understand traits and late static binding (http://php.net/manual/en/language.oop5.late-static-bindings.php).

```
<?php

$format = 'Class: %-13s | get_class(): %-13s | get_called_class(): %-13s%s';

trait TestTrait {
public function testMethod() {
global $format;
printf($format, __CLASS__, get_class(), get_called_class(), PHP_EOL);
}

public static function testStatic() {
global $format;
printf($format, __CLASS__, get_class(), get_called_class(), PHP_EOL);
}
}

trait DuplicateTrait {
public function duplMethod() {
global $format;
printf($format, __CLASS__, get_class(), get_called_class(), PHP_EOL);
}

public static function duplStatic() {
global $format;
printf($format, __CLASS__, get_class(), get_called_class(), PHP_EOL);
}
}

abstract class AbstractClass {

use DuplicateTrait;

public function absMethod() {
global $format;
printf($format, __CLASS__, get_class(), get_called_class(), PHP_EOL);
}

public static function absStatic() {
global $format;
printf($format, __CLASS__, get_class(), get_called_class(), PHP_EOL);
}
}

class BaseClass extends AbstractClass {
use TestTrait;
}

class TestClass extends BaseClass { }

$t = new TestClass();

$t->testMethod();
TestClass::testStatic();

$t->absMethod();
TestClass::absStatic();

$t->duplMethod();
```

```
TestClass::duplStatic();

?>

Will output:

Class: BaseClass | get_class(): BaseClass | get_called_class(): TestClass
Class: BaseClass | get_class(): BaseClass | get_called_class(): TestClass
Class: AbstractClass | get_class(): AbstractClass | get_called_class(): TestClass
Class: AbstractClass | get_class(): AbstractClass | get_called_class(): TestClass
Class: AbstractClass | get_class(): AbstractClass | get_called_class(): TestClass
Class: AbstractClass | get_class(): AbstractClass | get_called_class(): TestClass

Since Traits are considered literal "copying/pasting" of code, it's clear how the methods defined in DuplicateTrait give the same results as the methods defined in AbstractClass.
```

up
down
3

84td84 at gmail dot com ¶

8 years ago

```
A note to 'Beispiel #9 Statische Variablen'. A trait can also have a static property:

trait Counter {
static $trvar=1;

public static function stfunc() {
echo "Hello world!"
}
}

class C1 {
use Counter;
}

print "\nTRVAR: " . C1::$trvar . "\n"; //prints 1

$obj = new C1();
C1::stfunc(); //prints Hello world!
$obj->stfunc(); //prints Hello world!

A static property (trvar) can only be accessed using the classname (C1).
But a static function (stfunc) can be accessed using the classname or the instance ($obj).
```

up
down
5

artur at webprojektant dot pl ¶

11 years ago

```
Trait can not have the same name as class because it will show: Fatal error: Cannot redeclare class
```

up
down
3

Oddant ¶

10 years ago

```
I think it's obvious to notice that using 'use' followed by the traits name must be seen as just copying/pasting lines of code into the place where they are used.
```