




- [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

[Журнал изменений ООП »](#) [« Сериализация объектов](#)

- [Руководство по PHP](#)
- [Справочник языка](#)
- [Классы и объекты](#)

Change language: Russian

Ковариантность и контравариантность

В PHP 7.2.0 была добавлена частичная контравариантность путём устранения ограничений типа для параметров в дочернем методе. Начиная с PHP 7.4.0, добавлена полная поддержка ковариантности и контравариантности.

Ковариантность позволяет дочернему методу возвращать более конкретный тип, чем тип возвращаемого значения его родительского метода. В то время как контравариантность позволяет типу параметра в дочернем методе быть менее специфичным, чем в родительском.

Объявление типа считается более конкретным в следующем случае:

- Удалено [объединение типов](#)
- Добавлено [пересечение типов](#)
- Тип класса изменяется на тип дочернего класса
- [iterable](#) изменён на массив (array) или [Traversable](#)

В противном случае класс типа считается менее конкретным.

Ковариантность

Чтобы проиллюстрировать, как работает ковариантность, создадим простой абстрактный родительский класс *Animal*. *Animal* будет расширен за счёт дочерних классов *Cat* и *Dog*.

```
<?php
```

```
abstract class Animal
{
    protected string $name;

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

    abstract public function speak();
}

class Dog extends Animal
{
    public function speak()
    {
        echo $this->name . " лает";
    }
}

class Cat extends Animal
{
    public function speak()
    {
        echo $this->name . " мяукает";
    }
}
```

Обратите внимание, что в примере нет методов, которые возвращают значения. Будет добавлено несколько фабрик, которые возвращают новый объект типа класса *Animal*, *Cat* или *Dog*.

```
<?php
```

```
interface AnimalShelter
{
    public function adopt(string $name): Animal;
}
```

```

class CatShelter implements AnimalShelter
{
public function adopt(string $name): Cat // Возвращаем класс Cat вместо Animal
{
return new Cat($name);
}
}

class DogShelter implements AnimalShelter
{
public function adopt(string $name): Dog // Возвращаем класс Dog вместо Animal
{
return new Dog($name);
}
}

$kitty = (new CatShelter)->adopt("Рыжик");
$kitty->speak();
echo "\n";

$doggy = (new DogShelter)->adopt("Бобик");
$doggy->speak();

```

Результат выполнения приведённого примера:

```

Рыжик мяукает
Бобик лает

```

Контравариантность

В продолжение предыдущего примера, где мы использовали классы *Animal*, *Cat* и *Dog*, мы введём новые классы *Food* и *AnimalFood* и добавим в абстрактный класс *Animal* новый метод *eat(AnimalFood \$food)*.

```

<?php

class Food {}

class AnimalFood extends Food {}

abstract class Animal
{
protected string $name;

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

public function eat(AnimalFood $food)
{
echo $this->name . " ест " . get_class($food);
}
}

```

Чтобы увидеть суть контравариантности, мы переопределим метод *eat* класса *Dog* таким образом, чтобы он мог принимать любой объект класса *Food*. Класс *Cat* оставим без изменений.

```

<?php

class Dog extends Animal
{
public function eat(Food $food) {
echo $this->name . " ест " . get_class($food);
}
}

```

```
}  
}
```

Следующий пример покажет поведение контравариантности.

```
<?php  
  
$kitty = (new CatShelter)->adopt("Рыжик");  
$catFood = new AnimalFood();  
$kitty->eat($catFood);  
echo "\n";  
  
$doggy = (new DogShelter)->adopt("Бобик");  
$banana = new Food();  
$doggy->eat($banana);
```

Результат выполнения приведённого примера:

```
Рыжик ест AnimalFood  
Бобик ест Food
```

Но что случится, если *\$kitty* попытается съесть (**eat()**) банан (*\$banana*)?

```
$kitty->eat($banana);
```

Результат выполнения приведённого примера:

Fatal error: Uncaught TypeError: Argument 1 passed to Animal::eat() must be an instance of AnimalFood, instance of Food given

[+add a note](#)

User Contributed Notes 3 notes

[up](#)
[down](#)

86

[xedin dot unknown at gmail dot com ¶](#)

4 years ago

I would like to explain why covariance and contravariance are important, and why they apply to return types and parameter types respectively, and not the other way around.

Covariance is probably easiest to understand, and is directly related to the Liskov Substitution Principle. Using the above example, let's say that we receive an `AnimalShelter` object, and then we want to use it by invoking its `adopt()` method. We know that it returns an `Animal` object, and no matter what exactly that object is, i.e. whether it is a `Cat` or a `Dog`, we can treat them the same. Therefore, it is OK to specialize the return type: we know at least the common interface of any thing that can be returned, and we can treat all of those values in the same way.

Contravariance is slightly more complicated. It is related very much to the practicality of increasing the flexibility of a method. Using the above example again, perhaps the "base" method `eat()` accepts a specific type of food; however, a `_particular_` animal may want to support a `_wider range_` of food types. Maybe it, like in the above example, adds functionality to the original method that allows it to consume `_any_` kind of food, not just that meant for animals. The "base" method in `Animal` already implements the functionality allowing it to consume food specialized for animals. The overriding method in the `Dog` class can check if the parameter is of type `AnimalFood`, and simply invoke `parent::eat($food)`. If the parameter is `_not_` of the specialized type, it can perform additional or even completely different processing of that parameter - without breaking the original signature, because it `_still_` handles the specialized type, but also more. That's why it is also related closely to the Liskov Substitution: consumers may still pass a specialized food type to the `Animal` without knowing exactly whether it is a `Cat` or `Dog`.

[up](#)
[down](#)

9

[Anonymous ¶](#)

4 years ago

Covariance also works with general type-hinting, note also the interface:

```
interface xInterface  
{
```

```
public function y() : object;
}

abstract class x implements xInterface
{
abstract public function y() : object;
}
```

```
class a extends x
{
public function y() : \DateTime
{
return new \DateTime("now");
}
}
```

```
$a = new a;
echo '<pre>';
var_dump($a->y());
echo '</pre>';
```

[up](#)

[down](#)

3

[Hayley Watson ¶](#)

1 year ago

The gist of how the Liskov Substitution Principle applies to class types is, basically: "If an object is an instance of something, it should be possible to use it wherever an instance of something is allowed". The Co- and Contravariance rules come from this expectation when you remember that "something" could be a parent class of the object.

For the Cat/Animal example of the text, Cats are Animals, so it should be possible for Cats to go anywhere Animals can go. The variance rules formalise this.

Covariance: A subclass can override a method in the parent class with one that has a narrower return type. (Return values can be more specific in more specific subclasses; they "vary in the same direction", hence "covariant"). If an object has a method you expect to produce Animals, you should be able to replace it with an object where that method produces only Cats. You'll only get Cats from it but Cats are Animals, which are what you expected from the object.

Contravariance: A subclass can override a method in the parent class with one that has a parameter with a wider type. (Parameters can be less specific in more specific subclasses; they "vary in the opposite direction", hence "contravariant"). If an object has a method you expect to take Cats, you should be able to replace it with an object where that method takes any sort of Animal. You'll only be giving it Cats but Cats are Animals, which are what the object expected from you.

So, if your code is working with an object of a certain class, and it's given an instance of a subclass to work with, it shouldn't cause any trouble:

It might accept any sort of Animal where you're only giving it Cats, or it might only return Cats when you're happy to receive any sort of Animal, but LSP says "so what? Cats are Animals so you should both be satisfied."

[+add a note](#)

- [Классы и объекты](#)
 - [Введение](#)
 - [Основы](#)
 - [Свойства](#)
 - [Константы классов](#)
 - [Автоматическая загрузка классов](#)
 - [Конструкторы и деструкторы](#)
 - [Область видимости](#)
 - [Наследование](#)
 - [Оператор разрешения области видимости \(::\)](#)
 - [Ключевое слово static](#)
 - [Абстрактные классы](#)
 - [Интерфейсы объектов](#)
 - [Трейты](#)
 - [Анонимные классы](#)

- [Перегрузка](#)
- [Итераторы объектов](#)
- [Магические методы](#)
- [Ключевое слово final](#)
- [Клонирование объектов](#)
- [Сравнение объектов](#)
- [Позднее статическое связывание](#)
- [Объекты и ссылки](#)
- [Сериализация объектов](#)
- [Ковариантность и контравариантность](#)
- [Журнал изменений ООП](#)

- [Copyright © 2001-2024 The PHP Group](#)
- [My PHP.net](#)
- [Contact](#)
- [Other PHP.net sites](#)
- [Privacy policy](#)

