The Basics

- class
  - Readonly classes
- new
- Properties and methods
- extends
  - Signature compatibility rules
- ::class
- Nullsafe methods and properties

class

Basic class definitions begin with the keyword `class`, followed by a class name, followed by a pair of curly braces which enclose the definitions of the properties and methods belonging to the class.

The class name can be any valid label, provided it is not a PHP reserved word. As of PHP 8.4.0, using a single underscore `_` as a class name is deprecated. A valid class name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. As a regular expression, it would be expressed thus: `^[a-zA-Z_\x80-\xff][a-zA-Z0-9_\x80-\xff]*$`.

A class may contain its own constants, variables (called "properties"), and functions (called "methods").

Simple Class definition

```
<?php
class SimpleClass
{
    // property declaration
    public $var = 'a default value';
```

```
    // method declaration
    public function displayVar() {
        echo $this->var;
    }
}
?>
```

The pseudo-variable `$this` is available when a method is called from within an object context. `$this` is the value of the calling object.

> Calling a non-static method statically throws an `Error`. Prior to PHP 8.0.0, this would generate a deprecation notice, and `$this` would be undefined.
>
> Some examples of the `$this` pseudo-variable

```
<?php
class A
{
    function foo()
    {
        if (isset($this)) {
            echo '$this is defined (';
            echo get_class($this);
            echo ")\n";
        } else {
            echo "\$this is not defined.\n";
        }
    }
}

class B
{
    function bar()
    {
        A::foo();
    }
```

```
}

$a = new A();
$a->foo();

A::foo();

$b = new B();
$b->bar();

B::bar();
?>
```

ENTITY

```
$this is defined (A)

Deprecated: Non-static method A::foo() should not be called statically in %s  on line 27
$this is not defined.

Deprecated: Non-static method A::foo() should not be called statically in %s  on line 20
$this is not defined.

Deprecated: Non-static method B::bar() should not be called statically in %s  on line 32

Deprecated: Non-static method A::foo() should not be called statically in %s  on line 20
$this is not defined.
```

ENTITY

```
$this is defined (A)

Fatal error: Uncaught Error: Non-static method A::foo() cannot be called statically in
%s :27
Stack trace:
#0 {main}
  thrown in %s  on line 27
```

Readonly classes

As of PHP 8.2.0, a class can be marked with the readonly modifier. Marking a class as readonly will add the readonly modifier to every declared property, and prevent the creation of dynamic properties. Moreover, it is impossible to add support for them by using the `AllowDynamicProperties` attribute. Attempting to do so will trigger a compile-time error.

```
<?php
#[\AllowDynamicProperties]
readonly class Foo {
}

// Fatal error: Cannot apply #[AllowDynamicProperties] to readonly class Foo
?>
```

As neither untyped nor static properties can be marked with the `readonly` modifier, readonly classes cannot declare them either:

```
<?php
readonly class Foo
{
    public $bar;
}

// Fatal error: Readonly property Foo::$bar must have type
?>
```

```
<?php
readonly class Foo
{
    public static int $bar;
}

// Fatal error: Readonly class Foo cannot declare static properties
?>
```

A readonly class can be extended if, and only if, the child class is also a readonly class.

new

To create an instance of a class, the `new` keyword must be used. An object will always be created unless the object has a constructor defined that throws an Exception on error. Classes should be defined before instantiation (and in some cases this is a requirement).

If a variable containing a `string` with the name of a class is used with `new`, a new instance of that class will be created. If the class is in a namespace, its fully qualified name must be used when doing this.

> If there are no arguments to be passed to the class's constructor, parentheses after the class name may be omitted.

Creating an instance

```
<?php
class SimpleClass {
}

$instance = new SimpleClass();
var_dump($instance);

// This can also be done with a variable:
$className = 'SimpleClass';
$instance = new $className(); // new SimpleClass()
var_dump($instance);
?>
```

As of PHP 8.0.0, using `new` with arbitrary expressions is supported. This allows more complex instantiation if the expression produces a `string`. The expressions must be wrapped in parentheses.

Creating an instance using an arbitrary expression

In the given example we show multiple examples of valid arbitrary expressions that produce a class name. This shows a call to a function, string concatenation, and the `::class` constant.

```
<?php

class ClassA extends \stdClass {}
class ClassB extends \stdClass {}
class ClassC extends ClassB {}
class ClassD extends ClassA {}

function getSomeClass(): string
{
    return 'ClassA';
}

var_dump(new (getSomeClass()));
var_dump(new ('Class' . 'B'));
var_dump(new ('Class' . 'C'));
var_dump(new (ClassD::class));
?>
```

ENTITY

```
object(ClassA)#1 (0) {
}
object(ClassB)#1 (0) {
}
object(ClassC)#1 (0) {
}
object(ClassD)#1 (0) {
}
```

In the class context, it is possible to create a new object by `new self` and `new parent`.

When assigning an already created instance of a class to a new variable, the new variable will access the same instance as the object that was assigned. This behaviour is the same when passing instances to a function. A copy of an already created object can be made by

cloning it.

Object Assignment

```
<?php
class SimpleClass {
    public string $var;
}

$instance = new SimpleClass();

$assigned   =  $instance;
$reference  =& $instance;

$instance->var = '$assigned will have this value';

$instance = null; // $instance and $reference become null

var_dump($instance);
var_dump($reference);
var_dump($assigned);
?>
```

ENTITY

```
NULL
NULL
object(SimpleClass)#1 (1) {
  ["var"]=>
  string(30) "$assigned will have this value"
}
```

It's possible to create instances of an object in a couple of ways:

Creating new objects

```
<?php
```

```
class Test
{
    public static function getNew()
    {
        return new static();
    }
}

class Child extends Test {}

$obj1 = new Test(); // By the class name
$obj2 = new $obj1(); // Through the variable containing an object
var_dump($obj1 !== $obj2);

$obj3 = Test::getNew(); // By the class method
var_dump($obj3 instanceof Test);

$obj4 = Child::getNew(); // Through a child class method
var_dump($obj4 instanceof Child);

?>
```

ENTITY

```
bool(true)
bool(true)
bool(true)
```

It is possible to access a member of a newly created object in a single expression:

Access member of newly created object

```
<?php
echo (new DateTime())->format('Y'), PHP_EOL;

// surrounding parentheses are optional as of PHP 8.4.0
echo new DateTime()->format('Y'), PHP_EOL;
?>
```

ENTITY

```
2025
2025
```

> Prior to PHP 7.1, the arguments are not evaluated if there is no constructor function defined.

Properties and methods

Class properties and methods live in separate "namespaces", so it is possible to have a property and a method with the same name. Referring to both a property and a method has the same notation, and whether a property will be accessed or a method will be called, solely depends on the context, i.e. whether the usage is a variable access or a function call.

Property access vs. method call

```
<?php
class Foo
{
    public $bar = 'property';

    public function bar() {
        return 'method';
    }
}

$obj = new Foo();
echo $obj->bar, PHP_EOL, $obj->bar(), PHP_EOL;
```

ENTITY

```
property
method
```

That means that calling an anonymous function which has been assigned to a property is not directly possible. Instead the property has to be assigned to a variable first, for instance. It is possible to call such a property directly by enclosing it in parentheses.

Calling an anonymous function stored in a property

```
<?php
class Foo
{
    public $bar;

    public function __construct() {
        $this->bar = function() {
            return 42;
        };
    }
}

$obj = new Foo();

echo ($obj->bar)(), PHP_EOL;
```

ENTITY

```
42
```

extends

A class can inherit the constants, methods, and properties of another class by using the keyword `extends` in the class declaration. It is not possible to extend multiple classes; a class can only inherit from one base class.

The inherited constants, methods, and properties can be overridden by redeclaring them with the same name defined in the parent class. However, if the parent class has defined a method or constant as final, they may not be overridden. It is possible to access the overridden methods or static properties by referencing them with parent::.

> As of PHP 8.1.0, constants may be declared as final.

Simple Class Inheritance

```
<?php
class SimpleClass
{
    function displayVar()
    {
        echo "Parent class\n";
    }
}

class ExtendClass extends SimpleClass
{
    // Redefine the parent method
    function displayVar()
    {
        echo "Extending class\n";
        parent::displayVar();
    }
}

$extended = new ExtendClass();
$extended->displayVar();
?>
```

ENTITY

```
Extending class
Parent class
```

Signature compatibility rules

When overriding a method, its signature must be compatible with the parent method. Otherwise, a fatal error is emitted, or, prior to PHP 8.0.0, an `E_WARNING` level error is generated. A signature is compatible if it respects the variance rules, makes a mandatory parameter optional, adds only

optional new parameters and doesn't restrict but only relaxes the visibility. This is known as the Liskov Substitution Principle, or LSP for short. The constructor, and `private` methods are exempt from these signature compatibility rules, and thus won't emit a fatal error in case of a signature mismatch.

Compatible child methods

```
<?php

class Base
{
    public function foo(int $a) {
        echo "Valid\n";
    }
}

class Extend1 extends Base
{
    function foo(int $a = 5)
    {
        parent::foo($a);
    }
}

class Extend2 extends Base
{
    function foo(int $a, $b = 5)
    {
        parent::foo($a);
    }
}

$extended1 = new Extend1();
$extended1->foo();
$extended2 = new Extend2();
$extended2->foo(1);
```

ENTITY

```
Valid
Valid
```

The following examples demonstrate that a child method which removes a parameter, or makes an optional parameter mandatory, is not compatible with the parent method.

Fatal error when a child method removes a parameter

```
<?php

class Base
{
    public function foo(int $a = 5) {
        echo "Valid\n";
    }
}

class Extend extends Base
{
    function foo()
    {
        parent::foo(1);
    }
}
```

ENTITY

```
Fatal error: Declaration of Extend::foo() must be compatible with Base::foo(int $a = 5) in
/in/evtlq on line 13
```

Fatal error when a child method makes an optional parameter mandatory

```
<?php

class Base
{
    public function foo(int $a = 5) {
        echo "Valid\n";
```

```
    }
}

class Extend extends Base
{
    function foo(int $a)
    {
        parent::foo($a);
    }
}
```

ENTITY

```
Fatal error: Declaration of Extend::foo(int $a) must be compatible with Base::foo(int $a = 5)
in /in/qJXVC on line 13
```

Renaming a method's parameter in a child class is not a signature incompatibility. However, this is discouraged as it will result in a runtime `Error` if named arguments are used.

Error when using named arguments and parameters were renamed in a child class

```
<?php

class A {
    public function test($foo, $bar) {}
}

class B extends A {
    public function test($a, $b) {}
}

$obj = new B;

// Pass parameters according to A::test() contract
$obj->test(foo: "foo", bar: "bar"); // ERROR!
```

ENTITY

```
Fatal error: Uncaught Error: Unknown named parameter $foo in /in/XaaeN:14
Stack trace:
#0 {main}
  thrown in /in/XaaeN on line 14
```

::class

The `class` keyword is also used for class name resolution. To obtain the fully qualified name of a class `ClassName` use `ClassName::class`. This is particularly useful with namespaced classes.

Class name resolution

```
<?php
namespace NS {
    class ClassName {
    }

    echo ClassName::class;
}
?>
```

ENTITY

```
NS\ClassName
```

The class name resolution using `::class` is a compile time transformation. That means at the time the class name string is created no autoloading has happened yet. As a consequence, class names are expanded even if the class does not exist. No error is issued in that case.

Missing class name resolution

```
<?php
print Does\Not\Exist::class;
?>
```

ENTITY

```
Does\Not\Exist
```

As of PHP 8.0.0, `::class` may also be used on objects. This resolution happens at runtime, not compile time. Its effect is the same as calling `get_class` on the object.

Object name resolution

```
<?php
namespace NS {
    class ClassName {
    }

    $c = new ClassName();
    print $c::class;
}
?>
```

ENTITY

```
NS\ClassName
```

Nullsafe methods and properties

As of PHP 8.0.0, properties and methods may also be accessed with the "nullsafe" operator instead: `?->`. The nullsafe operator works the same as property or method access as above, except that if the object being dereferenced is ENTITY then ENTITY will be returned rather than an exception thrown. If the dereference is part of a chain, the rest of the chain is skipped.

The effect is similar to wrapping each access in an `is_null` check first, but more compact.

Nullsafe Operator

```
<?php

// As of PHP 8.0.0, this line:
$result = $repository?->getUser(5)?->name;

// Is equivalent to the following code block:
if (is_null($repository)) {
    $result = null;
} else {
    $user = $repository->getUser(5);
    if (is_null($user)) {
        $result = null;
    } else {
        $result = $user->name;
    }
}
?>
```

> The nullsafe operator is best used when null is considered a valid and expected possible value for a property or method return. For indicating an error, a thrown exception is preferable.

□□□ #2
□ □□□ □□□ 2026-10-31 17:39:57 UTC
□ □□□ □□□ 2026-10-31 17:39:59 UTC