# Scope Resolution Operator (::) ¶

The Scope Resolution Operator (also called Paamayim Nekudotayim) or in simpler terms, the double colon, is a token that allows access to static, constant, and overridden properties or methods of a class.

When referencing these items from outside the class definition, use the name of the class.

It's possible to reference the class using a variable. The variable's value can not be a keyword (e.g. `self`, `parent` and `static`).

Paamayim Nekudotayim would, at first, seem like a strange choice for naming a double-colon. However, while writing the Zend Engine 0.5 (which powers PHP 3), that's what the Zend team decided to call it. It actually does mean double-colon - in Hebrew!

**Example #1 :: from outside the class definition**

```
<?php
class MyClass {
const CONST_VALUE = 'A constant value';
}

$classname = 'MyClass';
echo $classname::CONST_VALUE;

echo MyClass::CONST_VALUE;
?>
```

Three special keywords *self*, *parent* and *static* are used to access properties or methods from inside the class definition.

**Example #2 :: from inside the class definition**

```
<?php
class OtherClass extends MyClass
{
```

```
public static $my_static = 'static var';

public static function doubleColon() {
echo parent::CONST_VALUE . "\n";
echo self::$my_static . "\n";
}
}

$classname = 'OtherClass';
$classname::doubleColon();

OtherClass::doubleColon();
?>
```

When an extending class overrides the parent's definition of a method, PHP will not call the parent's method. It's up to the extended class on whether or not the parent's method is called. This also applies to Constructors and Destructors, Overloading, and Magic method definitions.

**Example #3 Calling a parent's method**

```
<?php
class MyClass
{
protected function myFunc() {
echo "MyClass::myFunc()\n";
}
}

class OtherClass extends MyClass
{
// Override parent's definition
public function myFunc()
{
// But still call the parent function
parent::myFunc();
echo "OtherClass::myFunc()\n";
}
}

$class = new OtherClass();
```

```
$class->myFunc();
?>
```

See also some examples of static call trickery.

+ add a note

## User Contributed Notes 12 notes

up
down
209
***Theriault* ¶**
**13 years ago**

```
A class constant, class property (static), and class function (static) can all share the same name and be accessed using the double-colon.

<?php

class A {

    public static $B = '1'; # Static class variable.

    const B = '2'; # Class constant.

    public static function B() { # Static class function.
        return '3';
    }

}

echo A::$B . A::B . A::B(); # Outputs: 123
?>
```

up
down
94
***1naveengiri at gmail dot com* ¶**
**5 years ago**

```
In PHP, you use the self keyword to access static properties and methods.

The problem is that you can replace $this->method() with self::method() anywhere, regardless if method() is declared static or not. So which one
```

```
should you use?

Consider this code:

class ParentClass {
    function test() {
        self::who();    // will output 'parent'
        $this->who();    // will output 'child'
    }

    function who() {
        echo 'parent';
    }
}

class ChildClass extends ParentClass {
    function who() {
        echo 'child';
    }
}

$obj = new ChildClass();
$obj->test();
In this example, self::who() will always output ‘parent’, while $this->who() will depend on what class the object has.

Now we can see that self refers to the class in which it is called, while $this refers to the class of the current object.

So, you should use self only when $this is not available, or when you don’t want to allow descendant classes to overwrite the current method.
```

up
down
36
***guy at syntheticwebapps dot com ¶***
**9 years ago**

```
It seems as though you can use more than the class name to reference the static variables, constants, and static functions of a class definition
from outside that class using the :: . The language appears to allow you to use the object itself.

For example:
class horse
{
   static $props = {'order'=>'mammal'};
}
$animal = new horse();
echo $animal::$props['order'];

// yields 'mammal'

This does not appear to be documented but I see it as an important convenience in the language. I would like to see it documented and supported
as valid.

If it weren't supported officially, the alternative would seem to be messy, something like this:

$animalClass = get_class($animal);
echo $animalClass::$props['order'];
```

up
down
18
***jasverix at NOSPAM dot gmail dot com ¶***
**9 years ago**

```
Just found out that using the class name may also work to call similar function of anchestor class.

<?php

class Anchestor {

   public $Prefix = '';

   private $_string =  'Bar';
    public function Foo() {
        return $this->Prefix.$this->_string;
    }
}

class MyParent extends Anchestor {
    public function Foo() {
         $this->Prefix = null;
        return parent::Foo().'Baz';
    }
}
```

```
class Child extends MyParent {
    public function Foo() {
        $this->Prefix = 'Foo';
        return Anchestor::Foo();
    }
}

$c = new Child();
echo $c->Foo(); //return FooBar, because Prefix, as in Anchestor::Foo()

?>

The Child class calls at Anchestor::Foo(), and therefore MyParent::Foo() is never run.
```

up
down
9
***giovanni at gargani dot it ¶***
**13 years ago**

```
Well, a "swiss knife" couple of code lines to call parent method. The only limit is you can't use it with "by reference" parameters.
Main advantage you dont need to know the "actual" signature of your super class, you just need to know which arguments do you need

<?php
class someclass extends some superclass {
// usable for constructors
function __construct($ineedthisone) {
  $args=func_get_args();
  /* $args will contain any argument passed to __construct.
  * Your formal argument doesnt influence the way func_get_args() works
  */
  call_user_func_array(array('parent',__FUNCTION__),$args);
}
// but this is not for __construct only
function anyMethod() {
  $args=func_get_args();
  call_user_func_array(array('parent',__FUNCTION__),$args);
}
  // Note: php 5.3.0 will even let you do
function anyMethod() {
  //Needs php >=5.3.x
  call_user_func_array(array('parent',__FUNCTION__),func_get_args());
}

}
?>
```

up
down
10
***remy dot damour at ----no-spam---laposte dot net ¶***
**12 years ago**

As of php 5.3.0, you can use 'static' as scope value as in below example (add flexibility to inheritance mechanism compared to 'self' keyword...)

```
<?php

class A {
    const C = 'constA';
    public function m() {
        echo static::C;
    }
}

class B extends A {
    const C = 'constB';
}

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

// output: constB
?>
```

up
down
5
***wouter at interpotential dot com ¶***
**13 years ago**

It's worth noting, that the mentioned variable can also be an object instance. This appears to be the easiest way to refer to a static function as high in the inheritance hierarchy as possible, as seen from the instance. I've encountered some odd behavior while using static::something()

```
inside a non-static method.

See the following example code:

<?php
class FooClass {
    public function testSelf() {
        return self::t();
    }

    public function testThis() {
        return $this::t();
    }

    public static function t() {
        return 'FooClass';
    }

    function __toString() {
        return 'FooClass';
    }
}

class BarClass extends FooClass {
    public static function t() {
        return 'BarClass';
    }

}

$obj = new BarClass();
print_r(Array(
    $obj->testSelf(), $obj->testThis(),
));
?>

which outputs:

<pre>
```

```
Array
(
    [0] => FooClass
    [1] => BarClass
)
</pre>

As you can see, __toString has no effect on any of this. Just in case you were wondering if perhaps this was the way it's done.
```

up
down
3
***luka8088 at gmail dot com ¶***
**13 years ago**

```
Little static trick to go around php strict standards ...
Function caller founds an object from which it was called, so that static method can alter it, replacement for $this in static function but
without strict warnings :)

<?php

error_reporting(E_ALL + E_STRICT);

function caller () {
  $backtrace = debug_backtrace();
  $object = isset($backtrace[0]['object']) ? $backtrace[0]['object'] : null;
  $k = 1;

  while (isset($backtrace[$k]) && (!isset($backtrace[$k]['object']) || $object === $backtrace[$k]['object']))
    $k++;

  return isset($backtrace[$k]['object']) ? $backtrace[$k]['object'] : null;
}

class a {

  public $data = 'Empty';

  function set_data () {
    b::set();
  }
```

```
}

class b {

  static function set () {
    // $this->data = 'Data from B !';
    // using this in static function throws a warning ...
    caller()->data = 'Data from B !';
  }

}

$a = new a();
$a->set_data();
echo $a->data;

?>

Outputs: Data from B !

No warnings or errors !
```

up
down
2
***csaba dot dobai at php-sparcle dot com ¶***
**13 years ago**

```
For the 'late static binding' topic I published a code below, that demonstrates a trick for how to setting variable value in the late class, and
print that in the parent (or the parent's parent, etc.) class.

<?php

class cA
{
    /**
     * Test property for using direct default value
     */
    protected static $item = 'Foo';
    /**
     * Test property for using indirect default value
     */
    protected static $other = 'cA';

    public static function method()
    {
        print self::$item."\r\n"; // It prints 'Foo' on everyway... :(
        print self::$other."\r\n"; // We just think that, this one prints 'cA' only, but... :)
    }

    public static function setOther($val)
    {
        self::$other = $val; // Set a value in this scope.
    }
}

class cB extends cA
{
    /**
     * Test property with redefined default value
     */
    protected static $item = 'Bar';

    public static function setOther($val)
    {
        self::$other = $val;
    }
}

class cC extends cA
{
    /**
     * Test property with redefined default value
     */
    protected static $item = 'Tango';

    public static function method()
    {
        print self::$item."\r\n"; // It prints 'Foo' on everyway... :(
        print self::$other."\r\n"; // We just think that, this one prints 'cA' only, but... :)
    }

    /**
     * Now we drop redeclaring the setOther() method, use cA with 'self::' just for fun.
     */
}

class cD extends cA
{
    /**
     * Test property with redefined default value
     */
    protected static $item = 'Foxtrot';

    /**
     * Now we drop redeclaring all methods to complete this issue.
     */
}

cB::setOther('cB'); // It's cB::method()!
cB::method(); // It's cA::method()!
cC::setOther('cC'); // It's cA::method()!
cC::method(); // It's cC::method()!
cD::setOther('cD'); // It's cA::method()!
cD::method(); // It's cA::method()!

/**
* Results: ->
* Foo
* cB
* Tango
* cC
* Foo
* cD
*
* What the hell?! :)
*/
```

```
?>
```

up
down
0
***developit at mail dot ru ¶***
**16 years ago**

You use 'self' to access this class, 'parent' - to access parent class, and what will you do to access a parent of the parent? Or to access the very root class of deep class hierarchy? The answer is to use classnames. That'll work just like 'parent'. Here's an example to explain what I mean. Following code

```
<?php
class A
{
    protected $x = 'A';
    public function f()
    {
        return '['.$this->x.']';
    }
}

class B extends A
{
    protected $x = 'B';
    public function f()
    {
        return '{'.$this->x.'}';
    }
}

class C extends B
{
    protected $x = 'C';
    public function f()
    {
        return '('.$this->x.')'.parent::f().B::f().A::f();
    }
}
```

```
$a = new A();
$b = new B();
$c = new C();

print $a->f().'<br/>';
print $b->f().'<br/>';
print $c->f().'<br/>';
?>

will output

[A] -- {B} -- (C){C}{C}[C]
```

up
down
-1
***barss dot dev at gmail dot com ¶***
**14 years ago**

```
Nice trick with scope resolution
<?php
    class A
    {
        public function TestFunc()
        {
            return $this->test;
        }
    }

    class B
    {
        public $test;

        public function __construct()
        {
            $this->test = "Nice trick";
        }

        public function GetTest()
        {
            return A::TestFunc();
        }
    }

    $b = new B;
    echo $b->GetTest();
?>

will output

Nice trick
```

up

down

-16

***mongoose643 at gmail dot com ¶***

**15 years ago**

```
This is a solution for those that still need to write code compatible with php 4 but would like to use the flexibility of static variables. PHP 4 does not support static variables within the class scope but it does support them within the scope of class methods. The following is a bit of a workaround to store data in static mode in php 4.

Note: This code also works in PHP 5.

(Tested on version 4.3.1+)

The tricky part is when using when arrays you have to do a bit of fancy coding to get or set individual elements in the array. The example code below should show you the basics of it though.

<?php

class StaticSample
{
    //Copyright Michael White (www.crestidg.com) 2007
    //You may use and modify this code but please keep this short copyright notice in tact.
    //If you modify the code you may comment the changes you make and append your own copyright
    //notice to mine. This code is not to be redistributed individually for sale but please use it as part
    //of your projects and applications - free or non-free.


    //Static workaround for php4 - even works with arrays - the trick is accessing the arrays.
    //I used the format s_varname for my methods that employ this workaround. That keeps it
    //similar to working with actual variables as much as possible.
    //The s_ prefix immediately identifies it as a static variable workaround method while
    //I'm looking thorugh my code.
    function &s_foo($value=null, $remove=null)
    {
        static $s_var;    //Declare the static variable.    The name here doesn't matter - only the name of the method matters.

        if($remove)
        {
            if(is_array($value))
            {
                if(is_array($s_var))
                {
                    foreach($value as $key => $data)
                    {
                        unset($s_var[$key]);
                    }
                }
            }
            else
            {
                //You can't just use unset() here because the static state of the variable will bring back the value next time you call the
method.
                $s_var = null;
                unset($s_var);
            }
            //Make sure that you don't set the value over again.
            $value = null;
        }
        if($value)
        {
            if(is_array($value))
            {
                if(is_array($s_var))
                {
                    //$s_var = array_merge($s_var, $value);        //Doesn't overwrite values. This adds them - a property of the array_merge()
function.
                    foreach($value as $key => $data)
                    {
                        $s_var[$key] = $data;    //Overwrites values.
                    }
                }
                else
                {
                    $s_var = $value;
                }
            }
            else
            {
                $s_var = $value;
            }
        }

        return $s_var;
    }
}

echo "Working with non-array values.<br>";
echo "Before Setting: ".StaticSample::s_foo();
echo "<br>";
echo "While Setting: ".StaticSample::s_foo("VALUE HERE");
echo "<br>";
echo "After Setting: ".StaticSample::s_foo();
echo "<br>";
echo "While Removing: ".StaticSample::s_foo(null, 1);
echo "<br>";
echo "After Removing: ".StaticSample::s_foo();
echo "<hr>";
echo "Working with array values<br>";
$array = array(0=>"cat", 1=>"dog", 2=>"monkey");
echo "Set an array value: ";
print_r(StaticSample::s_foo($array));
echo "<br>";

//Here you need to get all the values in the array then sort through or choose the one(s) you want.
$all_elements = StaticSample::s_foo();
$middle_element = $all_elements[1];
echo "The middle element: ".$middle_element;
echo "<br>";

$changed_array = array(1=>"big dog", 3=>"bat", "bird"=>"flamingo");
echo "Changing the value: ";
print_r(StaticSample::s_foo($changed_array));
echo "<br>";

//All you have to do here is create an array with the keys you want to erase in it.
//If you want to erase all keys then don't pass any array to the method.
$element_to_erase = array(3=>null);
echo "Erasing the fourth element: ";
$elements_left = StaticSample::s_foo($element_to_erase, 1);
print_r($elements_left);
echo "<br>";
echo "Enjoy!";

?>
```

+ add a note