

Обектно-ориентирано програмиране

РНР 5

Съдържание

1. Основите.....	2
2. Автоматично зареждане на обекти.....	6
3. Конструктори и деструктори.....	7
4. Видимост на полета.....	9
5. Оператор за област на действие(::).....	12
6. Статични свойства и методи.....	14
7. Класови константи.....	16
8. Абстрактни класове.....	17
9. Интерфейси.....	19
10.Предефиниране.....	22
11.Итериране на обекти.....	27
12.Шаблони за дизайн.....	31
13.Вълшебни методи.....	33
14.Ключова дума final.....	36
15.Клониране на обекти.....	37
16.Сравняване на обекти.....	40
17.Отражение.....	42
18.Подказване на тип.....	57
19.Късно статично свързване.....	59

Въведение

В PHP 5 има нов обектен модел. Управлението на обектите в PHP е изцяло преработено, с цел по-добра производителност и повече възможности.

Основите

class

Всяка дефиниция на клас започва с ключовата дума `class`, последвана от името на класа, което може да бъде всяко име, което не е *запазена* дума в PHP. Следват чифт фигурни скоби, между които се поставя дефиницията на свойствата и методите на класа. Псевдо-променливата `$this` е налична, когато се извиква дадено свойство или метод от тялото на класа. `$this` представлява референция към извикания обект (обикновено обекта, на който принадлежи метода, но може да бъде и друг обект, ако методът се извиква *статично* в контекста на вторичния обект). Това е илюстрирано в следните примери:

```
<?php
class A
{
    function foo()
    {
        if (isset($this)) {
            echo '$this е дефинирана (';
            echo get_class($this);
            echo ")\n";
        } else {
            echo "\$this не е дефинирана.\n";
        }
    }
}

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

$a = new A();
$a->foo();
A::foo();
$b = new B();
$b->bar();
```

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

Примерът по-горе ще изведе:

```
$this е дефинирана (a)  
$this не е дефинирана.  
$this е дефинирана (b)  
$this не е дефинирана.
```

Пример #1 Проста дефиниция на клас

```
<?php  
class SimpleClass  
{  
    // дефиниция на свойство  
    public $var = 'стойност по подразбиране';  
  
    // дефиниция на метод  
    public function displayVar() {  
        echo $this->var;  
    }  
}  
?>
```

Стойността по подразбиране трябва да е константен израз, не (примерно) променлива, метод на клас или извикване на функция.

Пример #2 Стойност по подразбиране за член на клас

```
<?php  
class SimpleClass  
{  
    // невалидни декларации на членове:  
    public $var1 = 'hello '.'world';  
    public $var2 = <<<EOD  
hello world  
EOD;  
    public $var3 = 1+2;  
    public $var4 = self::myStaticMethod();  
    public $var5 = $myVar;  
  
    // валидни декларации на членове:  
    public $var6 = myConstant;  
    public $var7 = self::classConstant;  
    public $var8 = array(true, false);  
}  
?>
```

За разлика от *heredoc*, *nowdoc* може да се използва в контекста на всякакви статични данни.

Пример #3 Пример със статични данни

```
<?php
class foo {
    // От PHP 5.3.0
    public $bar = <<<'EOT'
bar
EOT;
}
?>
```

Забележка: Поддръжката на *nowdoc* е добавена в PHP 5.3.0.

new

За да бъде създадена инстанция на клас, трябва да се създаде нов обект и да се присвои на променлива. Когато се създава обект, той винаги ще бъде присвоен на променливата, освен ако няма *конструктор*, който да хвърля *изключение* при грешка. Класовете трябва да бъдат дефинирани преди инстанциране (а в някои случаи това е задължително).

Пример #4 Създаване на инстанция

```
<?php
$instance = new SimpleClass();
?>
```

В контекста на клас може да се създаде нов обект посредством *new self* и *new parent*.

Когато се присвоява вече създадена инстанция на клас към нова променлива, новата променлива ще има достъп до същата инстанция като присвоения обект. Нещата стоят по същия начин и когато се предават инстанции към функции. Копие на вече създаден обект може да се създаде чрез *клонирание*.

Пример #5 Присвояване на обект

```
<?php
```

```

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

$instance->var = '$assigned ще приеме тази стойност';

$instance = null; // $instance и $reference приемат стойност null

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

```

Примерът по-горе ще изведе:

```

NULL
NULL
object(SimpleClass)#1 (1) {
    ["var"]=>
        string(30) "$assigned ще приеме тази стойност"
}

```

extends

Даден клас може да наследи свойства и методи от друг клас, чрез използването на ключовата дума `extends`. Не се поддържа множествено наследяване, т.е. даден клас може да има само един базов клас.

Всички наследени свойства и методи могат да бъдат дефинирани отново, чрез повторното им дефиниране със същото име, с което са били дефинирани в родителския клас. Изключение се явяват случаите, когато в родителския клас даден метод е дефиниран като *final*. Достъпът до повторно дефинираните свойства или до статичните методи на родителския клас се осъществява чрез ключовата дума *parent::*

Пример #6 Просто наследяване на клас

```

<?php
class ExtendClass extends SimpleClass
{
    // Повторно дефиниране на родителския метод
    function displayVar()
    {
        echo "Наследяване на клас\n";
        parent::displayVar();
    }
}

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

```

Примерът по-горе ще изведе:

Автоматично зареждане на обекти

Много разработчици на обектно-ориентирани приложения създават по един PHP файл за всеки клас. Едно от най-досадните неща е създаването на дълъг списък на файловете за включване в началото на всеки скрипт (по един за всеки клас).

В PHP 5 това вече не е необходимо. Можете да дефинирате функция `__autoload`, която се извиква автоматично, в случай че се опитате да използвате клас/интерфейс, който все още не е дефиниран. Извикването на тази функция е последната възможност на скриптовата машина да зареди този клас преди да бъде генерирана фатална грешка.

Забележка: Изключения, хвърлени от функцията `__autoload`, не могат да бъдат хванати в блока `catch`, в следствие на което се генерира фатална грешка.

Забележка: Автоматичното зареждане не е достъпно при използване на PHP в интерактивен режим CLI.

Забележка: Ако се използва име на клас като например при `call_user_func()`, то може да съдържа някои опасни знаци като `../`. Препоръчително е да не използвате данни предоставени от потребителя в такива функции или поне да го проверявате във функцията `__autoload()`.

Пример #1 Пример за автоматично зареждане

В този пример се прави опит да се заредят класовете `MyClass1` и `MyClass2` от файловете `MyClass1.php` и съответно - `MyClass2.php`.

```
<?php  
function __autoload($class_name) {
```

```
require_once $class_name . '.php';
}
$obj = new MyClass1();
$obj2 = new MyClass2();
?>
```

Пример #2 Друг пример за автоматично зареждане

В този пример се прави опит за зареждане на интерфейс *ITest*.

```
<?php
function __autoload($name) {
    var_dump($name);
}
class Foo implements ITest {
}
/*
string(5) "ITest"
Fatal error: Interface 'ITest' not found in ...
*/
?>
```

Конструктори и деструктори

Конструктор

```
void __construct ([ mixed $args [, $... ] ] )
```

PHP 5 позволява на разработчиците да декларират конструктори на класовете. Класове, които имат метод-конструктор, извикват този метод при всяко създаване на нов обект, така че той е много подходящ за извършване на инициализации, от които обектът се нуждае преди да бъде използван.

Забележка: Конструкторите на родителските класове не се извикват автоматично, ако в даден дъщерен клас е дефиниран конструктор. За да се изпълни конструкторът на родителския

клас, е необходимо да се извика **parent::__construct()** в тялото на конструктора на дъщерния клас.

Пример #1 Използване на нови, унифицирани конструктори

```
<?php
class BaseClass {
    function __construct() {
        print "In BaseClass constructor\n";
    }
}

class SubClass extends BaseClass {
    function __construct() {
        parent::__construct();
        print "In SubClass constructor\n";
    }
}

$obj = new BaseClass();
$obj = new SubClass();
?>
```

Поради обратната съвместимост, ако PHP 5 не може да намери в даден клас метод **__construct()**, той ще потърси конструктор дефиниран по начина, по който се дефинира конструктор в по-старите версии, т.е. чрез функция с името на самия клас. На практика това означава, че може да възникнат проблеми при съвместимостта, единствено ако класът е имал метод **__construct()**, който е имал друг смисъл.

Деструктор

`void __destruct (void)`

В PHP 5 е въведен деструкторен метод, по подобие на другите обектно-ориентирани езици като C++. Деструкторът ще се извика в момента, в който обектът бъде унищожен.

Пример #2 Пример за деструктор

```
<?php
class MyDestructableClass {
    function __construct() {
        print "In constructor\n";
        $this->name = "MyDestructableClass";
    }
}
```

```
function __destruct() {
    print "Destroying " . $this->name . "\n";
}

$obj = new MyDestructableClass();
?>
```

Също както в случая с конструктора и тук деструкторът на родителския клас няма да бъде извикан автоматично. За да се изпълни деструкторът на родителския клас е необходимо изрично да се извика **parent::__destruct()** в тялото на деструктора.

Забележка: Деструкторите се извикват по време на спирането на скрипта, при което HTTP заглавките са изпратени. Работната директория по време на процеса на спирането на скрипта може да бъдат различна при някои SAPI-та (като например Apache).

Забележка: Опитът да се хвърли изключение в тялото на деструктор (извикан по време на приключване на работа на скрипта) ще предизвика фатална грешка.

Видимост на полета

Видимостта на свойство или метод може да бъде дефинирана, чрез използване на ключовите думи `public` (общодостъпен), `protected` (защитен) или `private` (частен). Полетата декларирани като `public` са достъпни от всякъде. Полетата `protected` могат да бъдат достъпни от дъщерните и родителските класове (и в класа, в който са дефинирани). `Private` позволява достъпа само от тялото на класа, в който е дефинирано полето.

Видимост на свойствата

Свойствата на даден клас трябва да бъдат декларирани като `public`, `private` или `protected`.

Пример #1 Декларация на свойство

```

<?php
/**
 * Дефиниция на MyClass
 */
class MyClass
{
    public $public = 'Public';
    protected $protected = 'Protected';
    private $private = 'Private';

    function printHello()
    {
        echo $this->public;
        echo $this->protected;
        echo $this->private;
    }
}

$obj = new MyClass();
echo $obj->public; // Работи
echo $obj->protected; // Fatal Error
echo $obj->private; // Fatal Error
$obj->printHello(); // Извежда Public, Protected и Private

/**
 * Дефиниция на MyClass2
 */
class MyClass2 extends MyClass
{
    // Може да се дефинират повторно public и protected методи, но не и private
    protected $protected = 'Protected2';

    function printHello()
    {
        echo $this->public;
        echo $this->protected;
        echo $this->private;
    }
}

$obj2 = new MyClass2();
echo $obj2->public; // Работи
echo $obj2->private; // Undefined
echo $obj2->protected; // Fatal Error
$obj2->printHello(); // Извежда Public, Protected2, Undefined
?>

```

Забележка: Ключавата дума *var* все още се поддържа от съображения за съвместимост (като синоним на ключовата дума *public*). В PHP 5 преди 5.1.3 употребата ѝ ще генерира предупреждение от тип **E_STRICT**.

Видимост на методи

Методите на даден клас трябва да бъдат декларирани като public, private или protected. Методи без такава декларация ще бъдат дефинирани като public.

Пример #2 Декларация на методи

```
<?php
/**
 * Дефиниция на MyClass
 */
class MyClass
{
    // Деклариране на public конструктор
    public function __construct() { }

    // Деклариране на public метод
    public function MyPublic() { }

    // Деклариране на protected метод
    protected function MyProtected() { }

    // Деклариране на private метод
    private function MyPrivate() { }

    // public метод
    function Foo()
    {
        $this->MyPublic();
        $this->MyProtected();
        $this->MyPrivate();
    }
}

$myclass = new MyClass;
$myclass->MyPublic(); // Работи
$myclass->MyProtected(); // Fatal Error
$myclass->MyPrivate(); // Fatal Error
$myclass->Foo(); // Извикват се Public, Protected и Private

/**
 * Дефиниция на MyClass2
 */
class MyClass2 extends MyClass
{
    // public метод
    function Foo2()
    {
        $this->MyPublic();
        $this->MyProtected();
        $this->MyPrivate(); // Fatal Error
    }
}
```

```

$myclass2 = new MyClass2;
$myclass2->MyPublic(); // Works
$myclass2->Foo2(); // Извикват се Public и Protected, но не и Private

class Bar
{
    public function test() {
        $this->testPrivate();
        $this->testPublic();
    }

    public function testPublic() {
        echo "Bar::testPublic\n";
    }

    private function testPrivate() {
        echo "Bar::testPrivate\n";
    }
}

class Foo extends Bar
{
    public function testPublic() {
        echo "Foo::testPublic\n";
    }

    private function testPrivate() {
        echo "Foo::testPrivate\n";
    }
}

$myFoo = new foo();
$myFoo->test(); // Bar::testPrivate
                // Foo::testPublic
?>

```

Оператор за област на действие (::)

Операторът за област на действие (също така, познат като Raamayim Nekudotayim) или просто двойно двоеточие, е символ, който предоставя достъп до константи, статични и повторно-дефинирани свойства и методи на даден клас.

Когато се обръщате към тези полета извън дефиницията на съответния клас, използвайте името на класа.

От PHP 5.3.0 е възможно да се обърнете към клас посредством променлива. Стойността на променливата не може да бъде ключова дума (напр. *self*, *parent* и *static* не са позволени при динамичните референции към класове.

В началото Раамауим Некудотайим може да ви се стори доста странен избор за име на двойното двоеточие. Докато се пишеше Zend Engine 0.5 (който е в основата на PHP 3), екипът на Zend реши да го кръсти така. Всъщност, това означава двойно двоеточие на иврит.

Пример #1 :: извън дефиницията на клас

```
<?php
class MyClass {
    const CONST_VALUE = 'Стойност на константата';
}

$classname = 'MyClass';
echo $classname::CONST_VALUE; // От PHP 5.3.0

echo MyClass::CONST_VALUE;
?>
```

Съществуват две специални ключови думи - *self* и *parent*, които се използват за достъп до свойства или методи вътре в дефиницията на класа.

Пример #2 :: в дефиницията на клас

```
<?php
class OtherClass extends MyClass
{
    public static $my_static = 'статична променлива';

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

$classname = 'OtherClass';
echo $classname::doubleColon(); // От PHP 5.3.0

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

Когато в дъщерен клас се дефинира повторно метод от родителски клас, PHP няма да извика родителския метод. Това дали да се извика родителския метод или не, се решава от дъщерния клас. Същото важи и за дефинициите на Конструкторите и деструкторите, Повторно-дефинираните и Вълшебните методи.

Пример #3 Извикване на метод на родителски клас

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

class OtherClass extends MyClass
{
    // Повторно дефиниране на метод на родителския клас
    public function myFunc()
    {
        // Извикване на метода на родителския клас
        parent::myFunc();
        echo "OtherClass::myFunc()\n";
    }
}

$class = new OtherClass();
$class->myFunc();
?>
```

Статични свойства и методи

Декларирането на свойствата и методите на клас като статични, ги прави достъпни без нуждата от инстанциране на класа. До поле, декларирано като статично, не може да се осъществи достъп от инстанция на обект (може само чрез статичен метод).

От съображения за съвместимост с PHP 4, ако не е използвана декларация за видимост, свойството или методът ще бъдат разглеждани като *public*.

Поради факта, че статичните методи могат да бъдат извикани без да е нужна инстанция на обект, псевдо-променливата *\$this* не е достъпна в метода деклариран като статичен.

Не е възможно да се осъществи достъп до статичните свойства на обект чрез оператора `->`.

Също както всяка статична променлива в PHP, статичните свойства могат да бъдат инициализирани само посредством низ или константа; изрази не са позволени. Така че, статично свойство може да бъде инициализирано примерно с цяло число или масив, но не може да бъде инициализирано с друга променлива, с върнатата стойност от функция или с обект.

При статичното извикване на нестатичен метод се генерира предупреждение от ниво E_STRICT.

От PHP 5.3.0 е възможно да се обърнете към клас посредством променлива. Стойността на променливата не може да бъде ключова дума (напр. *self*, *parent* и *static* не са позволени при динамичните референции към класове.

Пример #1 Пример за статично свойство

```
<?php
class Foo
{
    public static $my_static = 'foo';

    public function staticValue() {
        return self::$my_static;
    }
}

class Bar extends Foo
{
    public function fooStatic() {
        return parent::$my_static;
    }
}

print Foo::$my_static . "\n";

$foo = new Foo();
print $foo->staticValue() . "\n";
print $foo-
>my_static . "\n";          // Notice: Undefined property: Foo::$my_static

o::$my_static . "\n";
$classname = 'Foo';
print $classname::$my_static . "\n"; // От PHP 5.3.0

print Bar::$my_static . "\n";
$bar = new Bar();
print $bar->fooStatic() . "\n";
?>
```

Пример #2 Пример за статичен метод

```
<?php
class Foo {
```

```
public static function aStaticMethod() {
    // ...
}

Foo::aStaticMethod();
$classname = 'Foo';
$classname::aStaticMethod(); // От PHP 5.3.0
?>
```

Класови константи

В контекста на даден клас могат да бъдат дефинирани константи, които остават едни и същи и не търпят промяна. Константите се различават от нормалните променливи по това, че не е нужно да използвате символа \$, за да ги дефинирате или използвате.

Стойността трябва да бъде константен израз, т.е. не може да бъде променлива, член на клас, резултат от математическа операция или обръщане към функция.

Интерфейсите също могат да имат *константи*. За да видите някои примери прегледайте [документацията за интерфейсите](#).

От PHP 5.3.0 е възможно да се обърнете към клас посредством променлива. Стойността на променливата не може да бъде ключова дума (напр. *self*, *parent* и *static* не са позволени при динамичните референции към класове).

Пример #1 Дефиниране и използване на константа

```
<?php
class MyClass
{
    const constant = 'стойност на константата';

    function showConstant() {
        echo self::constant . "\n";
    }
}
```

```

echo MyClass::constant . "\n";

$classname = "MyClass";
echo $classname::constant . "\n"; // От PHP 5.3.0

$class = new MyClass();
$class->showConstant();
echo $class::constant . "\n"; // От PHP 5.3.0
?>

```

Пример #2 Пример със статични данни

```

<?php
class foo {
    // От PHP 5.3.0
    const bar = <<<'EOT'
bar
EOT;
}
?>

```

За разлика от heredoc, powdoc може да се използва в контекста на всякакви статични данни.

Забележка: Поддръжката на powdoc е добавена в PHP 5.3.0.

Абстрактни класове

В PHP 5 са реализирани абстрактни класове и методи. Не е възможно инстанцирането на клас дефиниран като абстрактен. Всеки клас, който съдържа поне един абстрактен метод, трябва също да се дефинира като абстрактен. Методите, деклариран като абстрактни, имат прототип, но не и имплементация.

При наследяване от абстрактен клас, всички методи деклариран като абстрактни в родителския клас трябва да бъдат дефинирани в дъщерния клас; освен това, тези методи трябва да бъдат със същата (или по-малко рестриктивна) видимост. Например ако абстрактният метод е дефиниран като protected, имплементацията на функцията трябва да бъде дефинирана като protected или public, но не и private.

Пример #1 Пример за абстрактни класове

```
<?php
abstract class AbstractClass
{
    // Методи, които трябва да бъдат дефинирани в дъщерния клас
    abstract protected function getValue();
    abstract protected function prefixValue($prefix);

    // Общ метод
    public function printOut() {
        print $this->getValue() . "\n";
    }
}

class ConcreteClass1 extends AbstractClass
{
    protected function getValue() {
        return "ConcreteClass1";
    }

    public function prefixValue($prefix) {
        return "{$prefix}ConcreteClass1";
    }
}

class ConcreteClass2 extends AbstractClass
{
    public function getValue() {
        return "ConcreteClass2";
    }

    public function prefixValue($prefix) {
        return "{$prefix}ConcreteClass2";
    }
}

$class1 = new ConcreteClass1;
$class1->printOut();
echo $class1->prefixValue('FOO_') . "\n";

$class2 = new ConcreteClass2;
$class2->printOut();
echo $class2->prefixValue('FOO_') . "\n";
?>
```

Примерът по-горе ще изведе:

```
ConcreteClass1
FOO_ConcreteClass1
ConcreteClass2
FOO_ConcreteClass2
```

Стар код, в който няма потребителски-дефинирани функции или класове с името 'abstract', ще работи без да са нужни промени по него.

Интерфейси

Интерфейсите позволяват да се дефинират методите, които даден клас задължително трябва да реализира, без да се декларират самите тела на тези методи.

Интерфейсите се дефинират чрез ключовата дума `interface`, аналогично на обикновения клас, но без да се декларират телата на методите му.

Всички методи, дефинирани в даден интерфейс, трябва да бъдат `public`, поради спецификата на интерфейса.

implements

За да се укаже, че даден клас реализира определен интерфейс, се използва операторът *implements*. Всички методи на интерфейса трябва да бъдат реализирани в класа. Нереализирането на метод предизвиква Фатална грешка. Даден клас може да реализира и повече от един интерфейс, като имената на интерфейсите в този случай се разделят със запетая.

Забележка: Даден клас не може да реализира два интерфейса, ако те имат методи с еднакви имена, тъй като това води до неопределеност.

Забележка: Интерфейсите могат да бъдат разширявани, също както класовете, посредством оператора *extend*

Константи

Интерфейсите могат да имат константи. Интерфейсните константи работят точно както класовите константи. Те не могат да бъдат предефинирани от класа/интерфейса, който ги наследява.

Примери

Пример #1 Пример за interface

```
<?php
// Декларация на интерфейс 'iTemplate'
interface iTemplate
{
    public function setVariable($name, $var);
    public function getHtml($template);
}

// Реализиране на интерфейса
// Това ще работи
class Template implements iTemplate
{
    private $vars = array();

    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;
    }
}

// Това няма да работи
// Fatal error: Class BadTemplate contains 1 abstract methods
// and must therefore be declared abstract (iTemplate::getHtml)
class BadTemplate implements iTemplate
{
    private $vars = array();

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

Пример #2 Разширяеми интерфейси

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

interface b extends a
{
    public function baz(Baz $baz);
}

// Това ще работи
class c implements b
```

```

{
    public function foo()
    {
    }

    public function baz(Baz $baz)
    {
    }
}

// Това няма да работи и ще предизвика Фатална грешка
class d implements b
{
    public function foo()
    {
    }

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

```

Пример #3 Множествено наследяване на интерфейси

```

<?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 #4 Интерфейси с константи

```
<?php
interface a
{
    const b = 'Интерфейсна константа';
}

// Извежда: Интерфейсна константа
echo a::b;

// Това разбира се няма да работи, тъй като не е позволено
// предефинирането на константи. Концепцията е същата като
// при класовите константи
class b implements a
{
    const b = 'Класова константа';
}
?>
```

Вж. също оператора [instanceof](#).

Предефиниране

Предефинирането в PHP осигурява средства за динамично "създаване" на членове и методи. Динамичните елементи се обработват посредством вълшебни методи, които могат да бъдат добавени към даден клас, с цел извършване на различни операции.

Предефиниращите методи се извикват при осъществяване на достъп до членове и методи, които не са декларирани или не са видими в текущата област на действие. По-нататък в раздела ще се използват термините "недостъпни членове" и "недостъпни методи" за указване на това съчетание от декларация и видимост.

Всички предефиниращи методи трябва да се декларират като *public*.

Забележка: Параметрите на тези вълшебни методи не могат да бъдат предавани по референция.

Забележка: Интерпретирането на "предефинирането" в PHP е различно от това на повечето обектно-ориентирани езици. Предефинирането обикновено предоставя възможността да съществуват множество методи с едно и също име, но с различен брой и тип на параметрите

Дневник на промените

Версия	Описание
5.1.0	Добавени са <code>__isset()</code> и <code>__unset()</code> .
5.3.0	Добавен е <code>__callStatic()</code> . Добавено е предупреждение за прилагане на <code>public</code> видимост и нестатично деклариране.

Предефиниране на свойства

```
void __set ( string $name , mixed $value )
mixed __get ( string $name )
bool __isset ( string $name )
void __unset ( string $name )
```

`__set()` се изпълнява при запис на данни в недостъпни членове.

`__get()` се използва за четене на данни от недостъпни членове.

`__isset()` се задейства при извикване на `isset()` или `empty()` с недостъпни членове.

`__unset()` се извиква когато се използва `unset()` с недостъпни членове.

Параметърът `$name` указва името на члена, с който се работи. Параметърът `$value` на метода `__set()` указва стойността в която трябва да се установи члена `$name`.

Предефинирането на членове работи само в контекста на обект. Тези вълшебни методи няма да бъдат извикани в контекста на статично извикване, следователно тези методи не могат да бъдат декларирани като статични.

Пример #1 Пример за предефиниране с `__get`, `__set`, `__isset` и `__unset`

```
<?php
class MemberTest {
    /** Място за предефинирани данни. */
    private $data = array();
```

```

    /** Предефиниране не се извършва при деклариран членове. */
    public $declared = 1;

    /** Предефинирането задейства само при осъществяване на достъп извън т
ялото на класа. */
    private $hidden = 2;

    public function __set($name, $value) {
        echo "Установяване на '$name' в '$value'\n";
        $this->data[$name] = $value;
    }

    public function __get($name) {
        echo "Връщане на стойността на '$name'\n";
        if (array_key_exists($name, $this->data)) {
            return $this->data[$name];
        }

        $trace = debug_backtrace();
        trigger_error(
            'Недефинирано свойство посредством __get(): ' . $name .
            ' в ' . $trace[0]['file'] .
            ' на ред ' . $trace[0]['line'],
            E_USER_NOTICE);
        return null;
    }

    /** От PHP 5.1.0 */
    public function __isset($name) {
        echo "Установен ли е '$name'? \n";
        return isset($this->data[$name]);
    }

    /** От PHP 5.1.0 */
    public function __unset($name) {
        echo "Унищожаване на '$name'\n";
        unset($this->data[$name]);
    }

    /** Това не е вълшебен метод, просто е тук за пример. */
    public function getHidden() {
        return $this->hidden;
    }
}

echo "<pre>\n";

$obj = new MemberTest;

$obj->a = 1;
echo $obj->a . "\n\n";

var_dump(isset($obj->a));
unset($obj->a);
var_dump(isset($obj->a));
echo "\n";

echo $obj->declared . "\n\n";

```

```
echo "Нека да експериментираме с private свойство именувано 'hidden':\n";
echo "Private свойствата са видими в тялото на класа, така че, не се използва
__get()...\n";
echo $obj->getHidden() . "\n";
echo "Private свойствата не са видими извън тялото на класа, така че, се използва
__get()...\n";
echo $obj->hidden . "\n";
?>
```

Примерът по-горе ще изведе:

```
Установяване на 'a' в '1'
Вземане на стойността на 'a'
1
Установен ли е 'a'?
bool(true)
Унищожаване на 'a'
Установен ли е 'a'?
bool(false)
1
Нека да експериментираме с private свойство именувано 'hidden'
Private свойствата са видими в тялото на класа, така че, не се използва
__get()...
2
Private свойствата не са видими извън тялото на класа, така че, се
използва __get()...
Връщане на стойността на 'hidden'
Notice: Undefined property via __get(): hidden in <file> on line 70 in
<file> on line 29
```

Предефиниране на методи

mixed **__call** (string \$name , array \$arguments)

mixed **__callStatic** (string \$name , array \$arguments)

__call() се задейства при извикване на недостъпни методи в контекста на обект.

__callStatic() се задейства при извикване на недостъпни методи в контекста на статично извикване.

Параметърът *\$name* указва името на метода, който се извиква.

Параметърът *\$arguments* е масив, съдържащ параметрите, които са предадени към метода *\$name*.

Пример #2 Предефиниране на методи с **__call** и **__callStatic**

```

<?php
class MethodTest {
    public function __call($name, $arguments) {
        // Забележка: стойността на $name е чувствителна към регистъра.
        echo "Извикване на метод на обект '$name' "
            . implode(', ', $arguments). "\n";
    }

    /** От PHP 5.3.0 */
    public static function __callStatic($name, $arguments) {
        // Забележка: стойността на $name е чувствителна към регистъра.
        echo "Извикване на статичен метод '$name' "
            . implode(', ', $arguments). "\n";
    }
}

$obj = new MethodTest;
$obj->runTest('в контекста на обект');

MethodTest::runTest('в контекста на статично извикване'); // От PHP 5.3.0
?>

```

Примерът по-горе ще изведе:

```

Извикване на метод на обект 'runTest' в контекста на обект
Извикване на статичен метод 'runTest' в контекста на статично извикване

```

Итериране на обекти

PHP 5 предоставя възможност на обектите да бъдат дефинирани така, че да могат да бъдат итерирани, примерно с цикъл `foreach`. По подразбиране всички видими свойства ще бъдат използвани при итерирането.

Пример #1 Просто итериране на обект

```

<?php
class MyClass
{

```

```

public $var1 = 'стойност 1';
public $var2 = 'стойност 2';
public $var3 = 'стойност 3';

protected $protected = 'protected променлива';
private $private = 'private променлива';

function iterateVisible() {
    echo "MyClass::iterateVisible:\n";
    foreach($this as $key => $value) {
        print "$key => $value\n";
    }
}

$class = new MyClass();

foreach($class as $key => $value) {
    print "$key => $value\n";
}
echo "\n";

$class->iterateVisible();

?>

```

Примерът по-горе ще изведе:

```

var1 => стойност 1
var2 => стойност 2
var3 => стойност 3
MyClass::iterateVisible:
var1 => стойност 1
var2 => стойност 2
var3 => стойност 3
protected => protected променлива
private => private променлива

```

Както показва резултатът, цикълът `foreach` обхожда всички видими променливи, до които има достъп. Ако отидем една крачка по-напред, можем да реализираме един от вътрешните интерфейси на PHP 5 - *Iterator*. Реализирането на този интерфейс дава възможност на обекта сам да решава какво и как ще се итерира.

Пример #2 Итериране на обект, реализиращ интерфейса *Iterator*

```

<?php
class MyIterator implements Iterator
{
    private $var = array();

    public function __construct($array)
    {

```

```

        if (is_array($array)) {
            $this->var = $array;
        }
    }

    public function rewind() {
        echo "превъртане\n";
        reset($this->var);
    }

    public function current() {
        $var = current($this->var);
        echo "текущ: $var\n";
        return $var;
    }

    public function key() {
        $var = key($this->var);
        echo "ключ: $var\n";
        return $var;
    }

    public function next() {
        $var = next($this->var);
        echo "следващ: $var\n";
        return $var;
    }

    public function valid() {
        $var = $this->current() !== false;
        echo "валиден: {$var}\n";
        return $var;
    }
}

$values = array(1,2,3);
$it = new MyIterator($values);

foreach ($it as $a => $b) {
    print "$a: $b\n";
}
?>

```

Примерът по-горе ще изведе:

```
превъртане
текущ: 1
валиден: 1
текущ: 1
ключ: 0
0: 1
следващ: 2
текущ: 2
валиден: 1
текущ: 2
ключ: 1
1: 2
следващ: 3
текущ: 3
валиден: 1
текущ: 3
ключ: 2
2: 3
следващ:
текущ:
валиден:
```

Можете да дефинирате вашия клас така, че да не се налага да дефинирате всички функции от интерфейса *Iterator*, като реализирате интерфейса *IteratorAggregate* в PHP 5.

Пример #3 Итериране на обект, реализиращ *IteratorAggregate*

```
<?php
class MyCollection implements IteratorAggregate
{
    private $items = array();
    private $count = 0;

    // Задължителен за дефиниране метод на интерфейса IteratorAggregate
    public function getIterator() {
        return new MyIterator($this->items);
    }

    public function add($value) {
        $this->items[$this->count++] = $value;
    }
}

$coll = new MyCollection();
$coll->add('стойност 1');
$coll->add('стойност 2');
$coll->add('стойност 3');

foreach ($coll as $key => $val) {
    echo "ключ/стойност: [$key -> $val]\n\n";
}
?>
```

Примерът по-горе ще изведе:

```
превъртане
текущ: стойност 1
валиден: 1
текущ: стойност 1
ключ: 0
ключ/стойност: [0 -> стойност 1]
следващ: стойност 2
текущ: стойност 2
валиден: 1
текущ: стойност 2
ключ: 1
ключ/стойност: [1 -> стойност 2]
следващ: стойност 3
текущ: стойност 3
валиден: 1
текущ: стойност 3
ключ: 2
ключ/стойност: [2 -> стойност 3]
следващ:
текущ:
валиден:
```

Забележка: За повече примери относно итераторите погледнете Разширение SPL.

Шаблони за дизайн

Шаблоните за дизайн са начин да се опишат най-добрите практики и методи на проектиране. Те дават гъвкави решения на често срещани в програмирането проблеми.

Метод фабрика (Factory Method)

Шаблонът Фабрика позволява инстанцирането на обекти по време на изпълнение на програмата. Нарича се метод фабрика, защото е отговорен за "производството" на обектите. При параметризираният метод фабрика (Parameterized Factory) името на класа, който трябва да инстанцира се получава като аргумент.

Пример #1 Параметризиран метод фабрика (Parameterized Factory Method)

```
<?php
class Example
{
    // Параметризиран метод фабрика
    public static function factory($type)
    {
        if (include_once 'Drivers/' . $type . '.php') {
            $classname = 'Driver_' . $type;
            return new $classname;
        } else {
            throw new Exception ('Драйверът не е намерен');
        }
    }
}
?>
```

Дефинирането на този метод в класа позволява на драйверите да бъдат заредени при нужда. Ако класът *Example* беше клас за абстракция на бази от данни, зареждането на драйверите за *MySQL* и *SQLite* щеше да изглежда по следния начин:

```
<?php
// Зареждане на драйвера за MySQL
$mysql = Example::factory('MySQL');

// Зареждане на драйвера за SQLite
$sqlite = Example::factory('SQLite');
?>
```

Сек (Singleton)

Шаблонът Сек се използва тогава, когато трябва да има точно една инстанция на даден клас. Най-често срещаният пример за това е връзка към база от данни. Реализирането на този шаблон позволява на

програмиста да направи тази единствена инстанция лесно достъпна за много други обекти.

Пример #2 Сек (Singleton) метод

```
<?php
class Example
{
    // Съдържа единствената инстанция на класа
    private static $instance;

    // private конструктор; предотвратява директното инстанциране на класа
    private function __construct()
    {
        echo 'I am constructed';
    }

    // Сек (Singleton) метод
    public static function singleton()
    {
        if (!isset(self::$instance)) {
            $c = __CLASS__;
            self::$instance = new $c;
        }

        return self::$instance;
    }

    // Примерен метод
    public function bark()
    {
        echo 'Woof!';
    }

    // Не позволява на потребителя да клонира инстанцията
    public function __clone()
    {
        trigger_error('Clone is not allowed.', E_USER_ERROR);
    }
}
?>
```

Това позволява да бъде върната само една единствена инстанция на класа *Example*.

```
<?php
// При опитът да се инстанцира този клас ще възникне грешка, тъй като конструктора на класа е private
$test = new Example;

// Това винаги ще връща единствената инстанция на класа
$test = Example::singleton();
$test->bark();

// Това ще върне грешка от ниво E_USER_ERROR.
$test_clone = clone $test;
```

Вълшебни методи

Имената на функциите `__construct`, `__destruct` (Вж. също Конструктори и деструктори), `__call`, `__callStatic`, `__get`, `__set`, `__isset`, `__unset` (Вж. също Повторно дефиниране), `__sleep`, `__wakeup`, `__toString`, `__set_state` и `__clone` са вълшебни в класовете на PHP. Не може да създавате функции с тези имена във вашите класове, освен ако не ги използвате по вълшебното им предназначение.

Внимание

PHP запазва всички имена на функции, започващи с `__`, като вълшебни. Препоръчително е да не използвате имена на функции, които започват с `__` в PHP, освен ако не използвате някоя документирана вълшебна функционалност.

`__sleep` и `__wakeup`

`serialize()` проверява дали в класа ви има функция с вълшебното име `__sleep`. Ако открие такава, то тази функция се изпълнява преди всяка сериализация. Тя би могла да изчисти обекта и се очаква да върне масив с имената на всички променливи от този обект, които ще бъдат сериализирани. Ако методът не върне нищо, тогава се сериализира стойността **NULL** и се генерира грешка от тип `E_NOTICE`.

Предназначението на `__sleep` е да затвори всички връзки към бази от данни, които обектът може да има, да съхрани незаписаните данни или да изпълни други подобни изчистващи задачи. Също така функцията може да се използва и в случай, че имате много големи обекти, които няма нужда да бъдат съхранени изцяло.

Обратно - `unserialize()` проверява за наличието на функция с вълшебното име `__wakeup`. Ако тя съществува, функцията може да възстанови всички ресурси, които даден обект може да има.

Употребата на `__wakeup` има за цел да възстанови всички връзки към бази от данни, които може да са били прекъснати по време на сериализацията и за извършване на други повторно-инициализиращи задачи.

Пример #1 Заспиване (`sleep`) и събуждане (`wakeup`)

```
<?php
class Connection {
    protected $link;
    private $server, $username, $password, $db;

    public function __construct($server, $username, $password, $db)
    {
        $this->server = $server;
        $this->username = $username;
        $this->password = $password;
        $this->db = $db;
        $this->connect();
    }

    private function connect()
    {
        $this->link = mysql_connect($this->server, $this->username, $this->password);
        mysql_select_db($this->db, $this->link);
    }

    public function __sleep()
    {
        return array('server', 'username', 'password', 'db');
    }

    public function __wakeup()
    {
        $this->connect();
    }
}
?>
```

`__toString`

Методът `__toString` позволява на даден клас да реши как ще реагира, в случай че бъде преобразуван до низ.

Пример #2 Прост пример

```
<?php
// Дефиниция на клас
class TestClass
{
    public $foo;
```

```

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

public function __toString() {
    return $this->foo;
}
}

$class = new TestClass('Здравей');
echo $class;
?>

```

Примерът по-горе ще изведе:

```
Здравей
```

Трябва да се отбележи, че до PHP 5.2.0 методът `__toString` ще бъде извикан, само когато е директно комбиниран с `echo()` или `print()`. От PHP 5.2.0 се извиква в контекста на всеки низов тип (например при `printf()` с модификатора `%s`), но не и в контекста на други типове (например при модификатора `%d`). От PHP 5.2.0 при преобразуването на обекти в низ без метода `__toString` ще се генерира **E_RECOVERABLE_ERROR**

`__set_state`

Този статичен метод се извиква за класове, експортирани от функцията `var_export()` от PHP 5.1.0.

Единственият параметър на метода е масив, съдържащ експортирани свойства във вида `array('property' => value, ...)`.

Пример #3 Употреба на `__set_state` (от PHP 5.1.0)

```

<?php

class A
{
    public $var1;
    public $var2;

    public static function __set_state($an_array) // От PHP 5.1.0
    {
        $obj = new A;
        $obj->var1 = $an_array['var1'];
        $obj->var2 = $an_array['var2'];
        return $obj;
    }
}

$a = new A;

```

```

$a->var1 = 5;
$a->var2 = 'foo';

eval('$b = ' . var_export($a, true) . '); // $b = A::__set_state(array(
                                          //   'var1' => 5,
                                          //   'var2' => 'foo',
                                          // ));

var_dump($b);

?>

```

Примерът по-горе ще изведе:

```

object(A)#2 (2) {
  ["var1"]=>
  int(5)
  ["var2"]=>
  string(3) "foo"
}

```

Ключова дума final

В PHP 5 е въведена ключовата дума `final`, като използването ѝ пред дефиницията на метод от родителски клас предотвратява възможността този метод да бъде дефиниран повторно в дъщерен клас. Ако даден клас е дефиниран като `final`, то той не може да бъде наследяван.

Пример #1 Пример за final метод

```

<?php
class BaseClass {
    public function test() {
        echo "BaseClass::test() called\n";
    }

    final public function moreTesting() {
        echo "BaseClass::moreTesting() called\n";
    }
}

class ChildClass extends BaseClass {
    public function moreTesting() {
        echo "ChildClass::moreTesting() called\n";
    }
}

```

```
}  
}  
// Резултатът ще е Fatal error: Cannot override final method BaseClass::moreTesting()  
?>
```

Пример #2 Пример за final клас

```
<?php  
final class BaseClass {  
    public function test() {  
        echo "BaseClass::test() called\n";  
    }  
  
    // Тук няма значение дали методът ще е final или не  
    final public function moreTesting() {  
        echo "BaseClass::moreTesting() called\n";  
    }  
}  
  
class ChildClass extends BaseClass {  
}  
// Резултатът ще е Fatal error: Class ChildClass may not inherit from final  
class (BaseClass)  
?>
```

Клониране на обекти

Създаването на копие на обект с абсолютно идентични свойства не винаги е желаният вариант. Добър пример за необходимостта от копиране на конструкторите е ситуацията, в която имате обект, който представлява GTK прозорец и съдържа ресурсите на този GTK прозорец. Когато създадете копие на този обект, може да искате да създадете нов прозорец със същите свойства и новият обект да съдържа ресурсите на новия прозорец. Като друг пример може да послужи ситуацията, в която вашият обект използва референция към друг обект, който използва и когато създадете копие на родителския обект, искате да се създаде нова инстанция и на другия обект, така че и той да си има свое собствено копие.

Копие на обект се създава посредством ключовата дума clone (която извиква метода __clone() на обекта, ако е възможно). Методът __clone() не може да бъде извикан директно.

```
$copy_of_object = clone $object;
```

Когато се създаде копие на обекта, PHP5 ще създаде нова инстанция на обекта, с негово собствено копие на свойствата. Всички свойства, които са референции към други променливи ще си останат референции, т.е. няма да се извърши дълбочинно копиране. Ако е дефиниран метод __clone(), ще бъде извикан метода __clone() на новосъздадения обект, за да може в случай на нужда да се променят стойностите на някои свойства.

Пример #1 Клониране на обект

```
<?php
class SubObject
{
    static $instances = 0;
    public $instance;

    public function __construct() {
        $this->instance = ++self::$instances;
    }

    public function __clone() {
        $this->instance = ++self::$instances;
    }
}

class MyCloneable
{
    public $object1;
    public $object2;

    function __clone()
    {
        // Принуждава създаването на копие на $this-
        >object, в противен случай
        // ще сочи към същия обект.
        $this->object1 = clone $this->object1;
    }
}

$obj = new MyCloneable();

$obj->object1 = new SubObject();
$obj->object2 = new SubObject();

$obj2 = clone $obj;

print("Оригинален обект:\n");
print_r($obj);

print("Клониран обект:\n");
```

```
print_r($obj2);  
?>
```

Примерът по-горе ще изведе:

```
Оригинален обект:  
MyCloneable Object  
(  
    [object1] => SubObject Object  
        (  
            [instance] => 1  
        )  
    [object2] => SubObject Object  
        (  
            [instance] => 2  
        )  
)  
Клониран обект:  
MyCloneable Object  
(  
    [object1] => SubObject Object  
        (  
            [instance] => 3  
        )  
    [object2] => SubObject Object  
        (  
            [instance] => 2  
        )  
)
```

Сравняване на обекти

В PHP 5 сравняването на обекти е по-сложен процес, отколкото в PHP 4 и е в съответствие с това, което може да се очаква от един Обектно-ориентиран език (не че PHP 5 е такъв език).

При използването на оператора за сравнение (`==`), обектите се сравняват по прост начин, а именно: Две инстанции на обект са равни, ако имат едни и същи атрибути и стойности и са инстанции на един и същи клас.

От друга страна, при използването на оператора за идентичност (`===`), обектите са равни тогава, и само тогава, когато сочат към една и съща инстанция на един и същ клас.

Следващият пример ще изясни тези правила.

Пример #1 Пример за сравняване на обекти в PHP 5

```
<?php
function bool2str($bool)
{
    if ($bool === false) {
        return 'FALSE';
    } else {
        return 'TRUE';
    }
}

function compareObjects(&$o1, &$o2)
{
    echo 'o1 == o2 : ' . bool2str($o1 == $o2) . "\n";
    echo 'o1 != o2 : ' . bool2str($o1 != $o2) . "\n";
    echo 'o1 === o2 : ' . bool2str($o1 === $o2) . "\n";
    echo 'o1 !== o2 : ' . bool2str($o1 !== $o2) . "\n";
}

class Flag
{
    public $flag;

    function Flag($flag = true) {
        $this->flag = $flag;
    }
}

class OtherFlag
{
```

```

public $flag;

function OtherFlag($flag = true) {
    $this->flag = $flag;
}
}

$o = new Flag();
$p = new Flag();
$q = $o;
$r = new OtherFlag();

echo "Две инстанции на един и същи клас\n";
compareObjects($o, $p);

echo "\nДве референции към една и съща инстанция\n";
compareObjects($o, $q);

echo "\nИнстанции на два отделни класа\n";
compareObjects($o, $r);
?>

```

Примерът по-горе ще изведе:

```

Две инстанции на един и същ клас
o1 == o2 : TRUE
o1 != o2 : FALSE
o1 === o2 : FALSE
o1 !== o2 : TRUE
Две референции към една и съща инстанция
o1 == o2 : TRUE
o1 != o2 : FALSE
o1 === o2 : TRUE
o1 !== o2 : FALSE
Инстанции на два отделни класа
o1 == o2 : FALSE
o1 != o2 : TRUE
o1 === o2 : FALSE
o1 !== o2 : TRUE

```

Забележка: Разширенията могат да дефинират свои собствени правила за сравняване на обекти.

Отражение

- Въведение
- Интерфейсът Reflector
- Класът ReflectionException
- Класът ReflectionFunction
- Класът ReflectionParameter
- Класът ReflectionClass
- Класът ReflectionObject
- Класът ReflectionMethod
- Класът ReflectionProperty
- Класът ReflectionExtension
- Наследяване на класовете за отражение

Въведение

PHP 5 разполага с цялостен API интерфейс за работа с отражения, което добавя възможността за обратно инженерство (reverse engineering) на класове, интерфейси, функции и методи, както и разширения. Освен това API-интерфейсът за отражения предлага начини за извличане на документиращи коментари за функции, класове и методи.

API-интерфейсът за отражения е обектно-ориентирано разширение на Zend Engine, което се състои от следните класове:

```
<?php
class Reflection { }
interface Reflector { }
class ReflectionException extends Exception { }
class ReflectionFunction extends ReflectionFunctionAbstract implements Reflector { }
class ReflectionParameter implements Reflector { }
class ReflectionMethod extends ReflectionFunctionAbstract implements Reflector { }
class ReflectionClass implements Reflector { }
class ReflectionObject extends ReflectionClass { }
class ReflectionProperty implements Reflector { }
class ReflectionExtension implements Reflector { }
?>
```

Забележка: За повече информация относно тези класове, вижте следващите глави.

Нека разгледаме следния пример:

Пример #1 Проста употреба на API интерфейса за отражения

```
<?php  
Reflection::export(new ReflectionClass('Exception'));  
?>
```

Примерът по-горе ще изведе:

```

Class [ <internal> class Exception ] {
  - Constants [0] {
  }
  - Static properties [0] {
  }
  - Static methods [0] {
  }
  - Properties [6] {
    Property [ <default> protected $message ]
    Property [ <default> private $string ]
    Property [ <default> protected $code ]
    Property [ <default> protected $file ]
    Property [ <default> protected $line ]
    Property [ <default> private $trace ]
  }
  - Methods [9] {
    Method [ <internal> final private method __clone ] {
    }
    Method [ <internal, ctor> public method __construct ] {
      - Parameters [2] {
        Parameter #0 [ <optional> $message ]
        Parameter #1 [ <optional> $code ]
      }
    }
    Method [ <internal> final public method getMessage ] {
    }
    Method [ <internal> final public method getCode ] {
    }
    Method [ <internal> final public method getFile ] {
    }
    Method [ <internal> final public method getLine ] {
    }
    Method [ <internal> final public method getTrace ] {
    }
    Method [ <internal> final public method getTraceAsString ] {
    }
    Method [ <internal> public method __toString ] {
    }
  }
}

```

Reflector

Reflector е интерфейс, реализиран от всички класове от тип `Reflection`, които могат да бъдат експортирани.

```
<?php
interface Reflector
{
    public string __toString()
    public static string export()
}
?>
```

ReflectionException

ReflectionException наследява стандартното изключение `Exception` и се хвърля от API-интерфейса за отражения. Няма специфични методи и свойства.

ReflectionFunction

Класът **ReflectionFunction** позволява извличане на информация за функции.

```
<?php
class ReflectionFunction extends ReflectionFunctionAbstract implements Reflector
{
    final private __clone()
    public void __construct(string name)
    public string __toString()
    public static string export()
    public string getName()
    public bool isInternal()
    public bool isDisabled()
    public mixed getClosure() /* От PHP 5.3.0 */
    public bool isUserDefined()
    public string getFileName()
    public int getStartLine()
    public int getEndLine()
    public string getDocComment()
    public array getStaticVariables()
    public mixed invoke([mixed args [, ...]])
    public mixed invokeArgs(array args)
    public bool returnsReference()
    public ReflectionParameter[] getParameters()
    public int getNumberOfParameters()
    public int getNumberOfRequiredParameters()
}
?>
```

Родителският клас **ReflectionFunctionAbstract** има същите методи с изключение на `invoke()`, `invokeArgs()`, `export()` и `isDisabled()`.

Забележка: `getNumberOfParameters()` и `getNumberOfRequiredParameters()` бяха добавени в PHP 5.0.3, а `invokeArgs()` беше добавен в PHP 5.1.0.

За да извършите интроспекция на функция, първо трябва да направите инстанция на класа **ReflectionFunction**. След това може да извикате който и да е от горните методи на тази инстанция.

Пример #2 Употреба на класа ReflectionFunction

```
<?php
/**
 * Обикновен брояч
 *
 * @return int
 */
function counter()
{
    static $c = 0;
    return $c++;
}

// Създаване на инстанция на класа ReflectionFunction
$func = new ReflectionFunction('counter');

// Извеждане на основна информация
printf(
    "===> %s функция '%s'\n".
    "    декларирана в %s\n".
    "    на редове от %d до %d\n",
    $func->isInternal() ? 'Вътрешна' : 'Дефинирана от потребителя',
    $func->getName(),
    $func->getFileName(),
    $func->getStartLine(),
    $func->getEndline()
);

// Извеждане на документиращ коментар
printf("---> Документация:\n %s\n", var_export($func->getDocComment(), 1));

// Извеждане на статичните променливи, ако има такива
if ($statics = $func->getStaticVariables())
{
    printf("---> Статични променливи: %s\n", var_export($statics, 1));
}

// Извикване на функция
printf("---> Резултат от извикването: ");
var_dump($func->invoke());

// може да се използва метода export()
echo "\nReflectionFunction::export() резултат:\n";
echo ReflectionFunction::export('counter');
?>
```

Забележка: Методът `invoke()` приема произволен брой аргументи, които се предават към функцията точно както при `call_user_func()`.

ReflectionParameter

Класът **ReflectionParameter** позволява извличане на информация за параметрите на функции и методи.

```
<?php
class ReflectionParameter implements Reflector
{
    final private __clone()
    public void __construct(string function, string parameter)
    public string __toString()
    public static string export()
    public string getName()
    public bool isPassedByReference()
    public ReflectionClass getDeclaringClass()
    public ReflectionClass getClass()
    public bool isArray()
    public bool allowsNull()
    public bool isPassedByReference()
    public bool isOptional()
    public bool isDefaultValueAvailable()
    public mixed getDefaultValue()
    public int getPosition()
}
?>
```

Забележка: `getDefaultValue()`, `isDefaultValueAvailable()` и `isOptional()` бяха добавени в PHP 5.0.3, а `isArray()` беше добавен в PHP 5.1.0. `getDeclaringFunction()` и `getPosition()` бяха добавени в PHP 5.2.3.

За да извършите интроспекция на параметрите на функция, първо трябва да направите инстанция на класовете **ReflectionFunction** или **ReflectionMethod** и да използвате техните методи `getParameters()`, за да върнете масива от параметри.

Пример #3 Употреба на класа ReflectionParameter

```
<?php
function foo($a, $b, $c) { }
function bar(Exception $a, &$b, $c) { }
function baz(ReflectionFunction $a, $b = 1, $c = null) { }
function abc() { }

// Създаване на инстанция на ReflectionFunction
// с параметри от командния ред
$reflect = new ReflectionFunction($argv[1]);
```

```

echo $reflect;

foreach ($reflect->getParameters() as $i => $param) {
    printf(
        "-- Параметър #%d: %s {\n".
        "  Клас: %s\n".
        "  Позволява NULL: %s\n".
        "  Предаден по адрес: %s\n".
        "  Опционален?: %s\n".
        "}\n",
        $i, // $param->getPosition() може да се използва от PHP 5.2.3
        $param->getName(),
        var_export($param->getClass(), 1),
        var_export($param->allowsNull(), 1),
        var_export($param->isPassedByReference(), 1),
        $param->isOptional() ? 'да' : 'не'
    );
}
?>

```

ReflectionClass

Класът **ReflectionClass** позволява извличане на информация за класове и интерфейси.

```

<?php
class ReflectionClass implements Reflector
{
    final private __clone()
    public void __construct(string name)
    public string __toString()
    public static string export(mixed class, bool return)
    public string getName()
    public bool isInternal()
    public bool isUserDefined()
    public bool isInstantiable()
    public bool hasConstant(string name)
    public bool hasMethod(string name)
    public bool hasProperty(string name)
    public string getFileName()
    public int getStartLine()
    public int getEndLine()
    public string getDocComment()
    public ReflectionMethod getConstructor()
    public ReflectionMethod getMethod(string name)
    public ReflectionMethod[] getMethods()
    public ReflectionProperty getProperty(string name)
    public ReflectionProperty[] getProperties()
    public array getConstants()
    public mixed getConstant(string name)
    public ReflectionClass[] getInterfaces()
    public bool isInterface()
    public bool isAbstract()
    public bool isFinal()
    public int getModifiers()
    public bool isInstance(stdclass object)
    public stdclass newInstance(mixed args)
    public stdclass newInstanceArgs(array args)
}

```

```

public ReflectionClass getParentClass()
public bool isSubclassOf(ReflectionClass class)
public array getStaticProperties()
public mixed getStaticPropertyValue(string name [, mixed default])
public void setStaticPropertyValue(string name, mixed value)
public array getDefaultProperties()
public bool isIterateable()
public bool implementsInterface(string name)
public ReflectionExtension getExtension()
public string getExtensionName()
}
?>

```

Забележка: `hasConstant()`, `hasMethod()`, `hasProperty()`, `getStaticPropertyValue()` и `setStaticPropertyValue()` бяха добавени в PHP 5.1.0, а `newInstanceArgs()` беше добавен в PHP 5.1.3.

За да извършите интроспекция на клас, първо трябва да направите инстанция на **ReflectionClass**. След това може да извикате който и да е от горните методи на тази инстанция.

Пример #4 Употреба на класа ReflectionClass

```

<?php
interface Serializable
{
    // ...
}

class Object
{
    // ...
}

/**
 * Клас Counter
 */
class Counter extends Object implements Serializable
{
    const START = 0;
    private static $c = Counter::START;

    /**
     * Извикване на брояча
     *
     * @access public
     * @return int
     */
    public function count() {
        return self::$c++;
    }
}

// Създаване на инстанция на класа ReflectionClass

```

```

$class = new ReflectionClass('Counter');

// Извеждане на основна информация
printf(
    "===> %s%s%s %s '%s' [наследява %s]\n" .
    "    деклариран в %s\n" .
    "    на редове от %d да %d\n" .
    "    притежава модификатори %d [%s]\n",
    $class->isInternal() ? 'Вътрешен' : 'Дефиниран от потребителя',
    $class->isAbstract() ? 'абстрактен' : '',
    $class->isFinal() ? 'финален' : '',
    $class->isInterface() ? 'интерфейс' : 'клас',
    $class->getName(),
    var_export($class->getParentClass(), 1),
    $class->getFileName(),
    $class->getStartLine(),
    $class->getEndline(),
    $class->getModifiers(),
    implode(' ', Reflection::getModifierNames($class->getModifiers()))
);

// Извеждане на документиращ коментар
printf("----> Документация:\n %s\n", var_export($class-
>getDocComment(), 1));

// Извежда интерфейсите реализирани от този клас
printf("----> Реализира:\n %s\n", var_export($class->getInterfaces(), 1));

// Извежда константите на класа
printf("----> Константи: %s\n", var_export($class->getConstants(), 1));

// Извежда свойствата на класа
printf("----> Свойства: %s\n", var_export($class->getProperties(), 1));

// Извежда методите на класа
printf("----> Методи: %s\n", var_export($class->getMethods(), 1));

// Ако класът може да се инстанцира, създава инстанция
if ($class->isInstantiable() {
    $counter = $class->newInstance();

    echo '----> $counter е инстанция? ';
    echo $class->isInstance($counter) ? 'да' : 'не';

    echo "\n----> new Object() е инстанция? ";
    echo $class->isInstance(new Object()) ? 'да' : 'не';
}
?>

```

Забележка: Методът `newInstance()` приема произволен брой аргументи, които се предават към функцията точно както при `call_user_func()`.

Забележка: `$class = new ReflectionClass('Foo');` `$class->isInstance($arg)` е еквивалентно на `$arg instanceof Foo` или `is_a($arg, 'Foo')`.

ReflectionObject

Класът **ReflectionObject** позволява извършването на обратно инженерство върху обекти.

```
<?php
class ReflectionObject extends ReflectionClass
{
    final private __clone()
    public void __construct(mixed object)
    public string __toString()
    public static string export(mixed object, bool return)
}
?>
```

ReflectionMethod

Класът **ReflectionMethod** позволява извличане на информация за методи.

```
<?php
class ReflectionMethod extends ReflectionFunctionAbstract implements Reflec
tor
{
    public void __construct(mixed class, string name)
    public string __toString()
    public static string export(mixed class, string name, bool return)
    public mixed invoke(stdclass object [, mixed args [, ...]])
    public mixed invokeArgs(stdclass object, array args)
    public bool isFinal()
    public bool isAbstract()
    public bool isPublic()
    public bool isPrivate()
    public bool isProtected()
    public bool isStatic()
    public bool isConstructor()
    public bool isDestructor()
    public int getModifiers()
    public mixed getClosure() /* От PHP 5.3.0 */
    public ReflectionClass getDeclaringClass()

    // Наследени от ReflectionFunctionAbstract
    final private __clone()
    public string getName()
    public bool isInternal()
    public bool isUserDefined()
    public string getFileName()
    public int getStartLine()
    public int getEndLine()
    public string getDocComment()
    public array getStaticVariables()
    public bool returnsReference()
    public ReflectionParameter[] getParameters()
    public int getNumberOfParameters()
    public int getNumberOfRequiredParameters()
}
?>
```

За да извършите интроспекция на метод, първо трябва да направите инстанция на **ReflectionMethod**. След това може да извикате който и да е от горните методи на тази инстанция.

Пример #5 Употреба на класа ReflectionMethod

```
<?php
class Counter
{
    private static $c = 0;

    /**
     * Увеличава брояча
     *
     * @final
     * @static
     * @access public
     * @return int
     */
    final public static function increment()
    {
        return ++self::$c;
    }
}

// Създаване на инстанция на класа ReflectionMethod
$method = new ReflectionMethod('Counter', 'increment');

// Извеждане на основна информация
printf(
    "====> %s%s%s%s%s%s метод '%s' (който е %s)\n" .
    "    деклариран в %s\n" .
    "    на редове от %d до %d\n" .
    "    притежава модификатори %d[%s]\n",
    $method->isInternal() ? 'Вътрешен' : 'Дефиниран от потребителя',
    $method->isAbstract() ? ' абстрактен' : '',
    $method->isFinal() ? ' финален' : '',
    $method->isPublic() ? ' public' : '',
    $method->isPrivate() ? ' private' : '',
    $method->isProtected() ? ' protected' : '',
    $method->isStatic() ? ' статичен' : '',
    $method->getName(),
    $method->isConstructor() ? 'конструктор' : 'нормален метод',
    $method->getFileName(),
    $method->getStartLine(),
    $method->getEndline(),
    $method->getModifiers(),
    implode(' ', Reflection::getModifierNames($method-
>getModifiers()))
);

// Извеждане на документиращ коментар
printf("----> Документация:\n %s\n", var_export($method-
>getDocComment(), 1));

// Извежда статичните променливи ако съществуват
if ($statics= $method->getStaticVariables()) {
    printf("----> Статични променливи: %s\n", var_export($statics, 1));
}
```

```
// Извиква метода
printf("----> Резултат от извикването: ");
var_dump($method->invoke(NULL));
?>
```

Пример #6 Getting closure using ReflectionMethod class

```
<?php

class Example {
    static function printer () {
        echo "Hello World!\n";
    }
}

$class = new ReflectionClass('Example');
$method = $class->getMethod('printer');
$closure = $method->getClosure(); /* От PHP 5.3.0 */
$closure(); // Hello World!

?>
```

Забележка: Опитът да се извика `private`, `protected` или абстрактен метод ще доведе до генериране на изключение от метода `invoke()`.

Забележка: Както се вижда от примера по-горе, при статични методи, трябва да се предава `NULL` като първи аргумент на `invoke()`. При нестатични методи, се предава инстанцията на класа.

ReflectionProperty

Класът **ReflectionProperty** позволява извличане на информация за свойства.

```
<?php
class ReflectionProperty implements Reflector
{
    final private __clone()
    public void __construct(mixed class, string name)
    public string __toString()
    public static string export(mixed class, string name, bool return)
    public string getName()
    public bool isPublic()
    public bool isPrivate()
    public bool isProtected()
    public bool isStatic()
    public bool isDefault()
    public void setAccessible() /* От PHP 5.3.0 */
    public int getModifiers()
```

```

public mixed getValue(stdclass object)
public void setValue(stdclass object, mixed value)
public ReflectionClass getDeclaringClass()
public string getDocComment()
}
?>

```

Забележка: `getDocComment()` беше добавен в PHP 5.1.0.
`setAccessible()` беше добавен в PHP 5.3.0.

За да извършите интроспекция на свойство, първо трябва да направите инстанция на **ReflectionProperty**. След това може да извикате който и да е от горните методи на тази инстанция.

Пример #7 Употреба на класа ReflectionProperty

```

<?php
class String
{
    public $length = 5;
}

// Създаване на инстанция на класа ReflectionProperty
$prop = new ReflectionProperty('String', 'length');

// Извеждане на основна информация
printf(
    "===> %s%s%s%s свойството '%s' (което беше %s)\n" .
    "    притежава модификатори %s\n",
    $prop->isPublic() ? ' public' : '',
    $prop->isPrivate() ? ' private' : '',
    $prop->isProtected() ? ' protected' : '',
    $prop->isStatic() ? ' статичен' : '',
    $prop->getName(),
    $prop-
>isDefault() ? 'декларирано по време на компилация' : 'създадено по време н
а стартиране',
    var_export(Reflection::getModifierNames($prop->getModifiers()), 1)
);

// Създаване на инстанция на String
$obj = new String();

// Връщане на текущата стойност
printf("----> Стойността е: ");
var_dump($prop->getValue($obj));

// Променяне на стойността
$prop->setValue($obj, 10);
printf("----> Установяване а стойността в 10, новата стойност е: ");
var_dump($prop->getValue($obj));

// Dump на обекта
var_dump($obj);
?>

```

Забележка: Опитът да се вземе или установи стойността на `private` или `protected` свойство на клас ще предизвика генериране на изключение.

ReflectionExtension

Класът **ReflectionExtension** позволява извличане на информация за разширения. Можете да извлечете всички заредени разширения по време на изпълнение чрез `get_loaded_extensions()`.

```
<?php
class ReflectionExtension implements Reflector {
    final private __clone()
    public void __construct(string name)
    public string __toString()
    public static string export(string name, bool return)
    public string getName()
    public string getVersion()
    public ReflectionFunction[] getFunctions()
    public array getConstants()
    public array getINIEntries()
    public ReflectionClass[] getClasses()
    public array getClassNames()
    public string info()
}
?>
```

За да извършите интроспекция на разширение, първо трябва да направите инстанция на **ReflectionExtension**. След това може да извикате който и да е от горните методи на тази инстанция.

Пример #8 Употреба на класа ReflectionExtension

```
<?php
// Създаване на инстанция на класа ReflectionExtension
$ext = new ReflectionExtension('standard');

// Извеждане на основна информация
printf(
    "Наименование   : %s\n" .
    "Версия         : %s\n" .
    "Функции        : [%d] %s\n" .
    "Константи      : [%d] %s\n" .
    "INI стойности  : [%d] %s\n" .
    "Класове        : [%d] %s\n",
    $ext->getName(),
    $ext->getVersion() ? $ext->getVersion() : 'NO_VERSION',
    sizeof($ext->getFunctions()),
    var_export($ext->getFunctions(), 1),

    sizeof($ext->getConstants()),
    var_export($ext->getConstants(), 1),

    sizeof($ext->getINIEntries()),
```

```

var_export($ext->getINIEntries(), 1),

sizeof($ext->getClassNames()),
var_export($ext->getClassNames(), 1)
);
?>

```

Наследяване на класовете за отражение

В случай че искате да създадете специализирани версии на вградените класове (примерно за да оцветите кода HTML при експортиране посредством лесни за достъп свойства вместо чрез методи или чрез прилагането на методи от тип utility), можете спокойно да ги наследите.

Пример #9 Наследяване на вградени класове

```

<?php
/**
 * Клас My_Reflection_Method
 */
class My_Reflection_Method extends ReflectionMethod
{
    public $visibility = array();

    public function __construct($o, $m)
    {
        parent::__construct($o, $m);
        $this->visibility= Reflection::getModifierNames($this->getModifiers());
    }
}

/**
 * Демонстрационен клас #1
 */
class T {
    protected function x() {}
}

/**
 * Демонстрационен клас #2
 */
class U extends T {
    function x() {}
}

// Извеждане на информацията
var_dump(new My_Reflection_Method('U', 'x'));
?>

```

Забележка: Внимание: Ако предефинирате конструктор, не забравяйте да извикате конструктора на родителския клас преди всякакъв друг код. В противен случай резултатът ще е следния:

Fatal error: Internal error: Failed to retrieve the reflection object

Подсказване на тип

В PHP 5 е въведено подсказване на тип. Функциите вече могат да принуждават параметрите да бъдат обекти (като се укаже името на класа преди наименованието на параметъра в дефиницията) или масиви (от PHP 5.1). Все пак, ако като стойност по подразбиране на параметъра се използва NULL, ще може да се използва като аргумент при всяко последващо извикване.

Пример #1 Примери за подсказване на тип

```
<?php
// Примерен клас
class MyClass
{
    /**
     * Тестова функция
     *
     * Първият параметър трябва да бъде обект от тип OtherClass
     */
    public function test(OtherClass $otherclass) {
        echo $otherclass->var;
    }

    /**
     * Друга тестова функция
     *
     * Първият параметър трябва да бъде масив
     */
    public function test_array(array $input_array) {
        print_r($input_array);
    }
}

// Друг примерен клас
class OtherClass {
    public $var = 'Hello World';
}
```

```
?>
```

Ако параметърът не удовлетворява подсказването за типа, изпълнението ще прекъсне с Фатална грешка, която може да бъде прихваната.

```
<?php
// Създаване на инстанция за всеки клас
$myclass = new MyClass;
$otherclass = new OtherClass;

// Fatal Error: Argument 1 must be an object of class OtherClass
$myclass->test('hello');

// Fatal Error: Argument 1 must be an instance of OtherClass
$foo = new stdClass;
$myclass->test($foo);

// Fatal Error: Argument 1 must not be null
$myclass->test(null);

// Работи: Извежда Hello World
$myclass->test($otherclass);

// Fatal Error: Argument 1 must be an array
$myclass->test_array('a string');

// Работи: Извежда масива
$myclass->test_array(array('a', 'b', 'c'));
?>
```

Подсказването на тип също така работи и с функции:

```
<?php
// Примерен клас
class MyClass {
    public $var = 'Hello World';
}

/**
 * Тестова функция
 *
 * Първият параметър трябва да бъде обект от тип MyClass
 */
function MyFunction (MyClass $foo) {
    echo $foo->var;
}

// Работи
$myclass = new MyClass;
MyFunction($myclass);
?>
```

Подсказване на типове с употреба на стойност NULL:

```
<?php
/* Присвояване на стойност NULL */
function test(stdClass $obj = NULL) {
```

```
}  
test(NULL);  
test(new stdClass);  
?>
```

Подсказването на типове може да бъде само за обекти и масиви (от PHP 5.1). Традиционното подсказване на тип с променливи от тип `int` и `string` не се поддържа.

Късно статично свързване

От PHP 5.3.0 в PHP е реализирано късно статично свързване, което може да се използва за обръщане към извикания клас в контекста на статичното наследяване.

Тази възможност неслучайно е наречена "късно статично свързване". "Късно свързване" идва от факта, че `static::` вече няма да сочи към класа, в който е дефиниран методът, а ще бъде изчисляван на базата на информацията по време на изпълнение на скрипта. Също така е наречено "статично свързване", тъй като може да се използва (без да е ограничено само в тази употреба) за извикване на статични методи.

Ограничения на `self::`

Статичните референции към текущия клас като `self::` и `__CLASS__` се извикват на базата на това на кой клас принадлежи методът, т.е. къде е дефиниран:

Пример #1 Употреба на `self::`

```
<?php  
class A {  
    public static function who() {  
        echo __CLASS__;  
    }  
    public static function test() {  
        self::who();  
    }  
}
```

```

    }
}

class B extends A {
    public static function who() {
        echo __CLASS__;
    }
}

B::test();
?>

```

Примерът по-горе ще изведе:

```
A
```

Употреба на късно статично свързване

С късното статично свързване се прави опит да се премахне това ограничение чрез въвеждането на нова ключова дума, която да сочи към класа, който първоначално е бил извикан по време на изпълнение. На практика това е ключовата дума, която би позволила да се извика *B* от *test()* в предишния пример. Веше решено вместо да се въвежда нова ключова дума, да се използва *static*, която така или иначе вече е запазена.

Пример #2 Проста употреба на *static::*

```

<?php
class A {
    public static function who() {
        echo __CLASS__;
    }
    public static function test() {
        static::who(); // Тук се извършва късното статично свързване
    }
}

class B extends A {
    public static function who() {
        echo __CLASS__;
    }
}

B::test();
?>

```

Примерът по-горе ще изведе:

```
B
```

Забележка: `static::` не работи еквивалентно на `$this` за статични методи! `$this->` следва правилата на наследяването, за разлика от `static::`. Тази разлика е обяснена по-подробно малко по-късно.

Пример #3 Употреба на `static::` в нестатичен контекст

```
<?php
class TestChild extends TestParent {
    public function __construct() {
        static::who();
    }

    public function test() {
        $o = new TestParent();
    }

    public static function who() {
        echo __CLASS__."\n";
    }
}

class TestParent {
    public function __construct() {
        static::who();
    }

    public static function who() {
        echo __CLASS__."\n";
    }
}

$o = new TestChild;
$o->test();

?>
```

Примерът по-горе ще изведе:

```
TestChild
TestParent
```

Забележка: Решението кой метод да се изпълни при късното статично свързване ще бъде взето при напълно определено статично извикване, без връщане. От друга страна, статичните извиквания посредством ключовите думи като `parent::` и `self::` ще препратят извикващата информация.

Пример #4 Препращащи и непрепращащи извиквания

```
<?php
class A {
    public static function foo() {
```

```

        static::who();
    }

    public static function who() {
        echo __CLASS__."\n";
    }
}

class B extends A {
    public static function test() {
        A::foo();
        parent::foo();
        self::foo();
    }

    public static function who() {
        echo __CLASS__."\n";
    }
}

class C extends B {
    public static function who() {
        echo __CLASS__."\n";
    }
}

C::test();
?>

```

Примерът по-горе ще изведе:

```

A
C
C

```

Крайни случаи

Съществуват редица различни начини да се извика метод в РНР, като например - обратни извиквания и вълшебни методи. Тъй като, при късното статично свързване, решението кой метод да се изпълни се базира на информацията по време на изпълнение, това може да доведе до неочаквани резултати при т.н. крайни случаи.

Пример #5 Късно статично свързване във вълшебни методи

```

<?php
class A {

    protected static function who() {
        echo __CLASS__."\n";
    }

    public function __get($var) {
        return static::who();
    }
}

```

```
}  
}  
  
class B extends A {  
    protected static function who() {  
        echo __CLASS__ . "\n";  
    }  
}  
  
$b = new B;  
$b->foo;  
?>
```

Примерът по-горе ще изведе:

```
B
```

Източник: <http://php.net/>