Оглавление

[**Основы ООП** 2](#_Toc429586574)

[**Классы и объекты в PHP** 4](#_Toc429586575)

[**Наследование классов в PHP** 10](#_Toc429586576)

[**Полиморфизм классов в PHP** 12](#_Toc429586577)

[**Работа с объектами классов PHP** 14](#_Toc429586578)

[**PHP5 и ООП** 16](#_Toc429586579)

[**PHP5 и ООП (Часть 2)** 20](#_Toc429586580)

[**PHP5 и ООП (Часть 3)** 24](#_Toc429586581)

[**PHP5 и ООП (Часть 4)** 27](#_Toc429586582)

[**Функции для работы с классами и объектами** 32](#_Toc429586583)

**Основы ООП**

В последнее время идея объектно-ориентированного программирования (ООП), кардинально новая идеология написания программ, все более занимает умы программистов.

Объектно-ориентированные программы более просты и мобильны, их легче модифицировать и сопровождать, чем их "традиционных" собратьев. Кроме того, похоже, сама идея объектной ориентированности при грамотном ее использовании позволяет программе быть даже более защищенной от различного рода ошибок, чем это задумывал программист в момент работы над ней. Однако ничего не дается даром: сами
идеи ООП довольно трудны для восприятия "с нуля", поэтому до сих пор очень большое количество программ (различные системы Unix, Apache, Perl, да и сам **PHP**) все еще пишутся на старом добром "объектно-неориентированном" Си.

PHP до недавнего времени обеспечивал лишь некоторую поддержку ООП. Однако, после выхода [**PHP5**](http://www.php.su/php/?php5) поддержка ООП в PHP стала практически полной.

Стратегию ООП лучше всего описать как смещение приоритетов в процессе программирования от функциональности приложения к структурам данных. Это позволяет программисту моделировать в создаваемых приложениях реальные объекты и ситуации. Технология ООП обладает тремя главными преимуществами:

* она проста для понимания: ООП позволяет мыслить категориями повседневных объектов;
* повышенно надежна и проста для сопровождения — правильное проектирование обеспечивает простоту расширения и модификации объектно-ориентированных программ. Модульная структура позволяет вносить независимые изменения в разные части программы, сводя к минимуму риск ошибок программирования;
* ускоряет цикл разработки — модульность и здесь играет важную роль, поскольку различные компоненты объектно-ориентированных программ можно легко использовать в других программах, что уменьшает избыточность кода и снижает риск внесения ошибок при копировании.

Специфика ООП заметно повышает эффективность труда программистов и позволяет им создавать более мощные, масштабируемые и эффективные приложения.

Объектно-ориентированное программирование основано на:

* Инкапсуляции;
* Полиморфизме;
* Наследовании.

**Инкапсуляция**

Инкапсуляция - это механизм, объединяющий данные и обрабатывающий их код как единое целое.

Многие преимущества ООП обусловлены одним из его фундаментальных принципов — инкапсуляцией. Инкапсуляцией называется включение различных мелких элементов в более крупный объект, в результате чего программист работает непосредственно с этим объектом. Это приводит к упрощению программы, поскольку из нее исключаются второстепенные детали.

Инкапсуляцию можно сравнить с работой автомобиля с точки зрения типичного водителя. Многие водители не разбираются в подробностях внутреннего устройства машины, но при этом управляют ею именно так, как было задумано. Пусть они не знают, как устроен двигатель, тормоз или рулевое управление, — существует специальный интерфейс, который автоматизирует и упрощает эти сложные операции. Сказанное также относится к инкапсуляции и ООП — многие подробности "внутреннего устройства" скрываются от пользователя, что позволяет ему сосредоточиться на решении конкретных задач. В ООП эта возможность обеспечивается [классами](http://www.php.su/learnphp/phpoo/?classes), [объектами](http://www.php.su/learnphp/phpoo/?classes) и различными средствами выражения иерархических связей между ними.

**Полиморфизм**

Полиморфизм позволяет использовать одни и те же имена для похожих, но технически разных задач. Главным в полиморфизме является то, что он позволяет манипулировать объектами путем создания стандартных интерфейсов для схожих действий. Полиморфизм значительно облегчает написание сложных программ.

**Наследование**

Наследование позволяет одному объекту приобретать свойства другого объекта, не путайте с копированием объектов. При копировании создается точная копия объекта, а при наследовании точная копия дополняется уникальными свойствами, которые характерны только для производного объекта.

**Классы и объекты в PHP**

**Класс** - это базовое понятие в объектно-ориентированном программировании ([ООП](http://www.php.su/learnphp/phpoo/?basic)). Классы образуют синтаксическую базу ООП. Их можно рассматривать как своего рода "контейнеры" для логически связанных данных и функций (обычно называемых методами — см. ниже). Если сказать проще, то класс - это своеобразный [тип данных](http://www.php.su/learnphp/datatypes/).

Экземпляр класса - это **объект**. Объект - это совокупность данных (свойств) и функций (методов) для их обработки. Свойства и методы называются членами класса. Вообще, объектом является все то, что поддерживает инкапсуляцию.

Если класс можно рассматривать как [тип данных](http://www.php.su/learnphp/datatypes/), то объект — как [переменную](http://www.php.su/learnphp/vars/) (по аналогии). Скрипт может одновременно работать с несколькими объектами одного класса, как с несколькими переменными.

Внутри объекта данные и код (члены класса) могут быть либо открыты, либо нет. Открытые данные и члены класса являются доступными для других частей программы, которые не являются частью объекта. А вот закрытые данные и члены класса доступны только внутри этого объекта.

Описание классов в PHP начинаются служебным словом **class**:

class Имя\_класса {
// описание членов класса - свойств и методов для их обработки
}

Для объявления объекта необходимо использовать оператор **new**:

Объект = new Имя\_класса;

Данные описываются с помощью служебного слова **var**. Метод описывается так же, как и обыкновенная [пользовательская функция](http://www.php.su/functions/custom/). Методу также можно передавать параметры.

Подведем промежуточные итоги: объявление класса должно начинаться с ключевого слова **class** (подобно тому, как объявление функции начинается с ключевого слова **function**). Каждому объявлению свойства, содержащегося в классе, должно предшествовать ключевое слово **var**. Свойства могут относиться к любому типу данных, поддерживаемых в РНР, их можно рассматривать как переменные с небольшими различиями. После объявлений свойств следуют объявления методов, очень похожие на типичные объявления пользовательских функций.

По общепринятым правилам имена классов ООП начинаются с прописной буквы, а все слова в именах методов, кроме первого, начинаются с прописных букв (первое слово начинается со строчной буквы). Разумеется, вы можете использовать любые обозначения, которые сочтете удобными; главное — выберите стандарт и придерживайтесь его.

Пример класса на PHP:

<?php
// Создаем новый класс Coor:
class Coor {
// данные (свойства):
var $name;
var $addr;

// методы:
 function Name() {
 echo "<h3>John</h3>";
 }

}

// Создаем объект класса Coor:
$object = new Coor;
?>

**Доступ к класам и объектам в PHP**

Мы рассмотрели, каким образом описываются классы и создаются объекты. Теперь нам необходимо получить доступ к членам класса, для этого в PHP предназначен оператор **->**. Приведем пример:

<?php
// Создаем новый класс Coor:
class Coor {
// данные (свойства):
var $name;

// методы:
 function Getname() {
 echo "<h3>John</h3>";
 }

}

// Создаем объект класса Coor:
$object = new Coor;
// Получаем доступ к членам класса:
$object->name = "Alex";
echo $object->name;
// Выводит 'Alex'
// А теперь получим доступ к методу класса (фактически, к функции внутри класса):
$object->Getname();
// Выводит 'John' заглавными буквами
?>

Чтобы получить доступ к членам класса внутри класса, необходимо использовать указатель **$this**, которы всегда относится к текущему объекту. Модифицированный метод **Getname()**:

function Getname() {
echo $this->name;
}

Таким же образом, можно написать метод **Setname()**:

function Setname($name) {
$this->name = $name;
}

Теперь для изменения имени можно использовать метод **Setname()**:

$object->Setname("Peter");
$object->Getname();

А вот и полный листинг кода:

<?php
// Создаем новый класс Coor:
class Coor {
// данные (свойства):
var $name;

// методы:
 function Getname() {
 echo $this->name;
 }

 function Setname($name) {
 $this->name = $name;
 }

}

// Создаем объект класса Coor:
$object = new Coor;
// Теперь для изменения имени используем метод Setname():
$object->Setname("Nick");
// А для доступа, как и прежде, Getname():
$object->Getname();
// Сценарий выводит 'Nick'
?>

Указатель **$this** можно также использовать для доступа к методам, а не только для доступа к данным:

function Setname($name) {
$this->name = $name;
$this->Getname();
}

**Конструкторы**

Довольно часто при создании объекта требуется задать значения некоторых свойств. К счастью, разработчики технологии ООП учли это обстоятельство и реализовали его в концепции конструкторов. Конструктор представляет собой метод, который задает значения некоторых свойств (а также может вызывать другие методы). Конструкторы вызываются автоматически при создании новых объектов. Чтобы это стало возможным, имя метода-конструктора должно совпадать с именем класса, в котором он содержится. Пример конструктора:

<?
class Webpage {
var $bgcolor;
 function Webpage($color) {
 $this->bgcolor = $color;
 }
}

// Вызвать конструктор класса Webpage
$page = new Webpage("brown");
?>

Раньше создание объекта и инициализация свойств выполнялись раздельно. Конструкторы позволяют выполнить эти действия за один этап.

Интересная подробность: в зависимости от количества передаваемых параметров могут вызываться разные конструкторы. В рассмотренном примере объекты класса *Webpage* могут создаваться двумя способами. Во-первых, вы можете вызвать конструктор, который просто создает объект, но не инициализирует его свойства:

$page = new Webpage;

Во-вторых, объект можно создать при помощи конструктора, определенного в классе, — в этом случае вы создаете объект класса Webpage и присваиваете значение его свойству *bgcolor*:

$page = new Webpage("brown");

**Деструкторы**

В РНР отсутствует непосредственная поддержка деструкторов. Тем не менее, вы можете легко имитировать работу деструктора, вызывая функцию РНР **unset()**. Эта функция уничтожает содержимое переменной и возвращает занимаемые ею ресурсы системе. С объектами **unset()** работает так же, как и с переменными. Допустим, вы работаете с объектом*$Webpage*. После завершения работы с этим конкретным объектом вызывается функция:

unset($Webpage);

Эта команда удаляет из памяти все содержимое *$Webpage*. Действуя в духе инкапсуляции, можно поместить вызов **unset()**в метод с именем *destroy()* и затем вызвать его:

$Website->destroy();

Необходимость в вызове деструкторов возникает лишь при работе с объектами, использующими большой объем ресурсов, поскольку все переменные и объекты автоматически уничтожаются по завершении сценария.

**Инициализация объектов**

Иногда возникает необходимость выполнить инициализацию объекта - присвоить его свойствам первоначальные значения. Предположим, имя класса *Coor* и он содержит два свойства:имя человека и город его проживания. Можно написать метод (функцию), который будет выполнять инициализацию объекта, например **Init()**:

<?php
// Создаем новый класс Coor:
class Coor {
// данные (свойства):
var $name;
var $city;

// Инициализирующий метод:
 function Init($name) {
 $this->name = $name;
 $this->city = "London";
 }

}

// Создаем объект класса Coor:
$object = new Coor;
// Для инициализации объекта сразу вызываем метод:
$object->Init();
?>

Главное не забыть вызвать функцию сразу после создания объекта, либо вызвать какой-нибудь метод между созданием (оператор **new**) объекта и его инициализацией (вызовом **Init**).

Для того, чтобы PHP знал, что определенный метод нужно вызывать автоматически при создании объекта, ему нужно дать имя такое же, как и у класса (**Coor**):

function Coor ($name)
$this->name = $name;
$this->city = "London";
}

Метод, инициализирующий объект, называется конструктором. Однако, PHP не имеет деструкторов, поскольку ресурсы освобождаюся автоматически при завершении работы скриптов.

**Обращение к элементам классов**

Обращение к элементам классов осуществляется с помощью оператора **::** "двойное двоеточие". Используя "двойное двоеточие", можно обращаться к методам классов.

При обращении к методам классов, программист должен использовать имена этих классов.

<?php
class A {
     function example() {
         echo "Это первоначальная функция A::example().<br>";
     }
}

class B extends A {
     function example() {
         echo "Это переопределенная функция B::example().<br>";
         A::example();
     }
}

// Не нужно создавать объект класса A.
// Выводит следующее:
// Это первоначальная функция A::example().
A::example();

// Создаем объект класса B.
$b = new B;

// Выводит следующее:
//   Это переопределенная функция B::example().
//   Это первоначальная функция A::example().
$b->example();
?>

В PHP5, используя эту лексему, программист может обращаться к константам, статическим или перегруженным свойствам или методам класса. Подробнее об этом [здесь](http://www.php.su/learnphp/phpoo/?php5_pm).

**Наследование классов в PHP**

**Наследование** - это не просто создание точной копии класса, а расширение уже существующего класса, чтобы потомок мог выполнять какие-нибудь новые, характерные только ему функции.

Итак, пусть у нас есть некоторый класс **A** с определенными свойствами и методами. Но то, что этот класс делает, нас не совсем устраивает — например, пусть он выполняет большинство функций, по сути нам необходимых, но не реализует некоторых других. Зададимся целью создать новый класс **B**, как бы "расширяющий" возможности класса **A**, добавляющий ему несколько новых свойств и методов. Сделать это можно двумя принципиально различными способами. Первый выглядит примерно так:

<?php
class A {
  function TestA() { ... }
  function Test() { ... }
}

class B {
  var $a; // объект класса A
  function B(параметры\_для\_A, другие\_параметры){
    $a = new A(параметры\_для\_A);
    // инициализируем другие поля B
  }
  function TestB() { ... }
  function Test() { ... }
}
?>

Поясним: в этой реализации объект класса **B** содержит в своем составе подобъект класса **A** в качестве свойства. Это свойство — лишь "частичка" объекта класса **B**, не более того. Подобъект не "знает", что он в действительности не самостоятелен, а содержится в классе **B**, поэтому не может предпринимать никаких действий, специфичных для этого класса.
Мы хотели получить расширение возможностей класса **A**, а не нечто, содержащее объекты **A**. Что означает "расширение"? Лишь одно: мы бы хотели, чтобы везде, где допустима работа с объектами класса **A**, была допустима и работа с объектами класса **B**. Но в приведенном примере это совсем не так.

Итак, мы имеем некоторые проблемы:

**1**. Мы не видим явно, что класс **B** лишь расширяет возможности **A**, а не является отдельной сущностью;
**2**. Мы должны обращаться к "части A" класса **B** через *$obj->a->TestA()*, а к членам самого класса **B** как *$obj->TestB()*. Последнее может быть довольно утомительным, если, как это часто бывает, в **B** будет использоваться очень много методов из **A** и гораздо меньше — из **B**. Кроме того, это заставляет нас постоянно помнить о внутреннем устройстве класса **B**.

Теперь на практике рассмотрим, что же представляет собой **наследование** (или расширение возможностей) классов:

<?php
class B extends A {
  function B(параметры\_для\_A, другие\_параметры) {
    $this->A(параметры\_для\_A);
    // инициализируем другие поля B
  }

  function TestB() { ... }
  function Test() { ... }
}
?>

Ключевое слово **extends** говорит о том, что создаваемый класс является лишь "расширением" класса **A**, и не более того. То есть **B** содержит те же самые свойства и методы, что и **A**, но, помимо них и еще некоторые дополнительные, "свои".
Теперь "часть A" находится прямо внутри класса **B** и может быть легко доступна, наравне с методами и свойствами самого класса **B**. Например, для объекта *$obj* класса **B** допустимы выражения *$obj->TestA()* и *$obj->TestB()*.

Итак, мы видим, что, действительно, класс **B** является воплощением идеи "расширение функциональности
класса **A**". Обратите также внимание: мы можем теперь забыть, что **B** унаследовал от **A** некоторые свойства или методы — снаружи все выглядит так, будто класс **B** реализует их самостоятельно.

Немного о терминологии: родительский класс **A** принято называть базовым классом, а класс дочерний класс**B** —производным от **A**. Иногда базовый класс также называют суперклассом, а производный — подкласcом.

Рассмотрим еще один пример на PHP:

<?php
class Parent {
  function parent\_funct() { echo "<h1>Это родительская функция</h1>"; }
  function test () { echo "<h1>Это родительский класс</h1>"; }
}

class Child extends Parent {
  function child\_funct() { echo "<h2>Это дочерняя функция</h2>"; }
  function test () { echo "<h2>Это дочерний класс</h2>"; }
}

$object = new Parent;
$object = new Child;

$object->parent\_funct(); // Выводит 'Это родительская функция'
$object->child\_funct(); // Выводит 'Это дочерняя функция'
$object->test(); // Выводит 'Это дочерний класс'
?>

Дочерний класс (подкласс) **Child** наследует все методы и свойства суперкласса **Parent**.

**Полиморфизм классов в PHP**

**Полиморфизм** (многоформенность) является следствием идеи наследования. В общих словах, полиморфность класса — это свойство базового класса использовать функции производных классов, даже если на момент определения еще неизвестно, какой именно класс будет включать его в качестве базового и, тем самым, становиться от него производным.

Рассмотрим свойство полиморфности классов на основе следующего примера:

<?php
class A {
// Выводит, функция какого класса была вызвана
function Test() { echo "Test from A\n"; }
// Тестовая функция — просто переадресует на Test()
function Call() { Test(); }
}
class B extends A {
// Функция Test() для класса B
function Test() { echo "Test from B\n"; }
}
$a=new A();
$b=new B();
?>

Используем следующие следующие команды:

$a->Call(); // выводит "Test from A"
$b->Test(); // выводит "Test from B"
$b->Call(); // Внимание! Выводит "Test from B"!

Обратите внимание на последнюю строчку: вопреки ожиданиям, вызывается не функция *Test()* из класса **A**, а функция из класса **B**! Складывается впечатление, что *Test()* из **B** просто переопределила функцию *Test()*из **A**. Так оно на самом деле и есть. Функция, переопределяемая в производном классе, называется виртуальной.

Механизм виртуальных функций позволяет, например, "подсовывать" функциям, ожидающим объект одного класса, объект другого, производного, класса. Еще один классический пример — класс, воплощающий собой свойства геометрической фигуры, и несколько производных от него классов — квадрат, круг, треугольник и т. д.
Базовый класс имеет виртуальную функцию *Draw()*, которая заставляет объект нарисовать самого себя. Все производные классы-фигуры, разумеется, переопределяют эту функцию (ведь каждую фигуру нужно рисовать по-особому). Также у нас есть массив фигур, причем мы не знаем, каких именно. Зато, используя полиморфизм, мы можем, не задумываясь, перебрать все элементы массива и вызвать для каждого из них метод *Draw()* — фигура сама "решит", какого она типа и как ее рисовать.

А вот еще один практический пример, показывающий свойство класса - полиморфизм:

<?php
class Base {
 function funct() {
 echo "<h2>Функция базового класса</h2>";
 }
 function base\_funct() {
 $this->funct();
 }
}

class Derivative extends Base {
 function funct() {
 echo "<h3>Функция производного класса</h3>";
 }
}

$b = new Base();
$d = new Derivative();

$b->base\_funct();
$d->funct();
$d->base\_funct();
// Скрипт выводит:

// Функция базового класса
// Функция производного класса
// Функция производного класса
?>

В рассмотренном примере функция **base\_funct()** класса **Base** была перезаписана одноименной функцией класса**Derivative**.

**Работа с объектами классов PHP**

**Копирование объектов**

Так уж устроен PHP, что в нем все переменные, в том числе и объекты, всегда рассматриваются как простой набор значений и копируются целиком. Например, если у нас есть объект **$a** и мы выполняем оператор **$b=$a**, то все содержимое **$a** будет скопировано в **$b** один-в-один.

<?php
class A {
// Создаем новый метод:
 function Test() {
 echo "<h1>Hello!</h1>";
 }
}
// Создаем объект класса A:
$a=new A();
// Копируем объект $a:
$b=$a;
// Теперь работаем с новым объектом $b
$b->Test(); // Выводит 'Hello!'
?>

**Сравнение объектов**

В PHP 4 объекты сравниваются очень просто: по именам. Два объекта равны, если они имеют те же самые свойства и значения, а также являются экземплярами одного и того же класса. Сравнение двух объектов осуществляют, используя оператор эквивалентности (**===**). Вот пример:

<?php
class A {
// Создаем новый метод:
 function Test() {
 echo "<h1>Hello!</h1>";
 }
}

// Создаем объект класса A:
$a=new A();
// Создаем объект класса A:
$b=new A();
// Выводит 'Объекты равны':
if ($a===$b) echo "<h3>Объекты равны</h2>";
?>

**Ссылки на объект**

PHP позволяет создавать ссылки на объекты. Вот пример:

<?php
class A {
// Создаем новый метод:
 function Test() {
 echo "<h1>Hello!</h1>";
 }
}

// Создаем объект класса A:
$a=new A();
// Ссылка на объект класса A:
$b=& new A();
$b->Test();
?>

Так как PHP5 является практически полноценным объектно-ориентированным языком, то в нем, соответственно, больше возможностей для работы с объектами. Ознакомиться с этими возможностями вы можете [здесь](http://www.php.su/learnphp/phpoo/?php5).

**PHP5 и ООП**

**Новая объектно-ориентированная модель в PHP5**

Когда Зив Сераски (Zeev Suraski) добавил объектно-ориентированный (ОО) синтаксис в PHP 3, это можно было рассматривать как "синтаксический подсластитель для поддержки классов" ("syntactic sugar for accessing collections"). Объектно-ориентированная модель получила поддержку наследования и позволяла классу (и объекту) объединять методы и свойства, но не более того. Когда Зив и Анди переписали движок для PHP 4, это был полностью новый движок, работающий намного быстрее, намного стабильнее и с еще многими другими возможностями. Однако, изменения практически не затронули ОО модель, первоначально введенную еще в РНР 3.

Хотя объектная модель имела серьезные ограничения, она широко использовалась, часто в очень больших приложениях, написанных на PHP. Это победное шествование парадигмы ООП, даже такой ограниченной в РНР 4, привело к тому, что изменения объектной модели стали центральными в новом релизе РНР - PHP5.

Какие были ограничения в PHP 3 и 4? Самым большим ограничением (которое и приводило ко всем остальным ограничениям) был тот факт, что семантика экземпляра объекта была такой же, что и для родных типов. Как это фактически отражалось на разработчиках? Когда вы присваивали переменную (которая указывает на объект) другой переменной, то создавалась копия объекта. Мало того, что это влияло на производительность, но и это обычно приводило к ошибкам в приложении, потому что многие разработчики думали, что обе переменные будут указывать на тот же самый объект. А они указывали на разные копии того же самого объекта, поэтому, изменяя один объект, мы не меняли другой. Вот пример:

<?php
class Person {
     var $name;
     function getName() {
         return $this->name;
     }
     function setName($name) {
         $this->name = $name;
     }
     function Person($name) {
         $this->setName($name);
     }
}

function changeName($person, $name) {
     $person->setName($name);
}

$person = new Person("Andi");
changeName($person, "Stig");
print $person->getName();
?>

В РНР 4 этот код выведет "Andi". Причина кроется в том, что мы передаем объект *$person* в функцию *changeName()* по значению, а не по ссылке, таким образом, объект *$person* будет скопирован, и *changeName()* будет работать уже с копией объекта *$person*.

Такое поведение не является интуитивно понятным. Действительно, многие разработчики ожидали Java-подобного поведения. В Java, переменные фактически являются указателями на объект, и поэтому при дублировании будет скопирован указатель, а не сам объект.

Было два вида разработчиков: те, кто знал об этой проблеме, и те, кто не знал. Последние, обычно, не сталкивались с этой проблемой, потому что их код был написан так, что было безразлично, существует ли такая проблема или нет. Конечно, некоторые из этих разработчиков проводили бессонные ночи в "увлекательных" поисках "сверхъестественных" ошибок. Первая группа также имела проблему, поскольку приходилось вручную определять передачу объекта по ссылке, запрещая движку копировать объекты, и код был испещрен многочисленными знаками '**&**'.

Старая объектная модель приводит не только к вышеупомянутым проблемам, но также вскрывает более фундаментальные проблемы, которые на существующей объектной модели не позволяли осуществлять другие возможности.

В PHP 5 объектная модель была полностью переписана для того, чтобы сразу работать с указателями на объект. Если вы явно не клонируете объект, используя ключевое слово **clone**, вы никогда не будете работать с копией объекта, думая, что работаете с самим объектом. В PHP 5 уже не нужно явно передавать объекты или присваивать их по ссылке, это делаетсяавтоматически.

Обратите внимание: явная передача и присваивание по ссылке также поддерживается, на тот случай, если вы хотите изменить содержимое переменной или объекта.

**Новый объектно-ориентированный подход в PHP5**

Новые возможности объектной модели являются слишком многочисленными. Приведем обзор главных изменений:

1. **public/private/protected - модификаторы доступа для методов и свойств**

Позволяют управлять доступом к методам и свойствам. Теперь видимость свойств и методов может быть определена ключевыми словами: **public**, **private**, **protected**. Модификатор **public** позволяет обращаться к свойствам и методам отовсюду. Модификатор **private** позволяет обращаться к свойствам и методам только внутри текущего класса. Модификатор **protected** позволяет обращаться к свойствам и методам только текущего класса и класса, который наследует свойства и методы текущего класса.

<?php
/\*\*
  \* Define 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; // Works
echo $obj->protected; // Fatal Error
echo $obj->private; // Fatal Error
$obj->printHello(); // Shows Public, Protected and Private

/\*\*
  \* Define MyClass2
  \*/
class MyClass2 extends MyClass
{
     // We can redeclare the public and protected method, but not private
     protected $protected = 'Protected2';

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

$obj2 = new MyClass2();
echo $obj->public; // Works
echo $obj2->private; // Undefined
echo $obj2->protected; // Fatal Error
$obj2->printHello(); // Shows Public, Protected2, not Private

?>

1. **Унифицированный конструктор \_\_construct()**

PHP 5 позволяет объявлять методы-конструкторы. Классы, в которых объявлен метод-констуктор, будут вызывать этот метод при каждом создании нового объекта, так что это может оказаться полезным, чтобы, например, инициализировать какое-либо состояние объекта перед его использованием. Конструктор, ранее совпадавший с названием класса, теперь необходимо объявлять как **\_\_construct()**, что позволит легче перемещать классы в иерархиях.Конструкторы в классах-родителях не вызываются автоматически. Чтобы вызвать конструктор, объявленный в родительском классе, следует обратиться к методу **parent::\_\_construct()**.

<?php
class BaseClass {
     function \_\_construct() {
         print "Конструктор класса BaseClass\n";
     }
}

class SubClass extends BaseClass {
     function \_\_construct() {
         parent::\_\_construct();
         print "Конструктор класса SubClass\n";
     }
}

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

Если PHP 5 не может обнаружить объявленный метод **\_\_construct()**, вызов конструктора произойдет по прежней схеме, через обращение к методу, имя которого соответствует имени класса. Может возникнуть только одна проблема совместимости старого кода, если в нём присутствуют классы с методами **\_\_construct()**.

1. **Поддержка деструктора для класса, определяемого как метод \_\_destructor()**

PHP 5 предоставляет концепцию деструкторов, сходную с теми, что применяются в других ОО языках, таких, как Java: когда освобождается последняя ссылка на объект, перед высвобождением памяти, занимаемой этим объектом, вызывается метод **\_\_destruct()**, не принимающий параметров.

<?php
class MyDestructableClass {
     function \_\_construct() {
         print "Конструктор\n";
         $this->name = "MyDestructableClass";
     }

     function \_\_destruct() {
         print "Уничтожается " . $this->name . "\n";
     }
}

$obj = new MyDestructableClass();
?>

Как и в случае с конструкторами, деструкторы, объявленные в родительском классе, не будут вызваны автоматически. Для вызова деструктора, объявленном в классе-родителе, следует обратиться к методу**parent::\_\_destruct()**.

**PHP5 и ООП (Часть 2)**

1. **Явное клонирование объекта**

Создание копии объекта с абсолютно идентичными свойствами не всегда является приемлемым вариантом. Например, когда ваш объект содержит ссылку на какой-либо другой используемый объект и, когда вы создаёте копию ссылающегося объекта, вам нужно также создать новый экземпляр содержащегося объекта, так, чтобы копия объекта содержала собственный отдельный экземпляр содержащегося объекта.

Копия объекта создается с использованием вызова **clone** (который вызывает метод **\_\_clone()** объекта, если это возможно). Вы можете объявить метод **\_\_clone()**, который будет вызван при клонировании объекта (после того, как все свойства будут скопированы из исходного объекта).

copy\_of\_object = clone $object;

Когда программист запрашивает создание копии объекта, PHP 5 определит, был ли для этого объекта объявлен метод **\_\_clone()** или нет. Если нет, будет вызван метод **\_\_clone()**, объявленный по умолчанию, который скопирует все свойства объекта. Если метод **\_\_clone()** был объявлен, создание копий свойств в копии объекта полностью возлагается на него. Для удобства, движок обеспечивает программиста функцией, которая импортирует все свойства из объекта-источника, так что программист может осуществить позначное копирование свойств и переопределять только необходимые. Приведем пример клонирования объекта:

<?php
class MyClass {
     function \_\_clone() {
         print "Объект был клонирован ";
     }
}
$obj = new MyClass();
clone $obj;
?>

1. **Константы класса**

В определения классов теперь можно включить константы, и ссылаться на них, используя объект. Константы также могут быть объявлены и в пределах одного класса. Отличие переменных и констант состоит в том, что при объявлении последних или при обращении к ним не используется символ *$*. Как и свойства и методы, значения констант, объявленных внутри класса, не могут быть получены через переменную, содержащую экземпляр этого класса.

<?php
class MyClass {
     const SUCCESS = "Success";
     const FAILURE = "Failure";
}
print MyClass::SUCCESS;
?>

1. **Статические члены класса**

Определения классов могут теперь включить статических членов класса (свойства и методы), доступ к которым осуществляется через класс. Общее использование статических членов показано на примере:

<?php
class Singleton {
     static private $instance = NULL;

     private function \_\_construct() {
     }

     static public function getInstance() {
         if (self::$instance == NULL) {
             self::$instance = new Singleton();
         }
         return self::$instance;
     }
}
?>

1. **Статические методы**

Вы можете теперь определить методы как статические, разрешая им быть вызванными вне контекста объекта. Статические методы не определяются через переменную **$this**, поскольку они не должны быть ограничены определенным объектом.

<?php
class MyClass {
     static function helloWorld() {
         print "Hello, world";
     }
}
MyClass::helloWorld();
?>

1. **Абстрактные классы**

PHP 5 поддерживает определение абстрактных классов и методов. Создавать экземпляр класса, который был объявлен абстрактным, нельзя. Класс, в котором объявлен хотя бы один абстрактный метод, должен также быть объявлен абстрактным. Методы, объявленные как абстрактные, несут, по существу, лишь описательный смысл и не могут включать какой-либо функционал. Класс может быть объявлен как абстрактный при помощи использования ключевого слова **abstract**, для исключения из обработки движком описания класса. Однако, вы можете наследовать абстрактные классы. Практический пример:

<?php

abstract class AbstractClass {

    /\* Данный метод должен быть определён в дочернем классе \*/
    abstract protected function getValue();

    /\* Общий метод \*/
    public function print() {
      print $this->getValue();
    }

}

class ConcreteClass1 extends AbstractClass {

    protected function getValue() {
      return "ConcreteClass1";
    }

}

class ConcreteClass2 extends AbstractClass {

    protected function getValue() {
      return "ConcreteClass2";
    }

}

$class1 = new ConcreteClass1;
$class1->print();

$class2 = new ConcreteClass2;
$class2->print();
?>

1. **Абстрактные методы**

Метод может быть объявлен как **abstract**, таким образом отложив его определение наследуемым классом. Класс, который включает абстрактные методы, должен быть объявлен как **abstract**.

<?php
abstract class MyBaseClass {
     abstract function display();
}
?>

1. **Указание класса как типа**

Определения функции могут включить указание типа класса, передаваемого в качестве параметра. Если функция будет вызвана с неправильным типом, произойдет ошибка.

<?php
function expectsMyClass(MyClass $obj) {

}
?>

1. **Поддержка разыменования объектов, которые возвращаются методами.**

В PHP 4 вы не могли непосредственно разыменовывать объекты, которые возвращаются из методов. Вы должны были бы сначала присвоить такой объект некой фиктивной переменной.

Поясним на примере. В PHP 4:

<?php
$dummy = $obj->method();
$dummy->method2();
?>

В PHP 5:

<?php
$obj->method()->method2();
?>

1. **Итераторы**

PHP 5 предоставляет механизм итераторов для получения списка всех свойств какого-либо объекта, например, для использования совместно с оператором *foreach*. По умолчанию, в итерации будут участвовать все свойства, объявленные как **public**. Пример использования итераторов:

<?php

class MyClass {
   public $var1 = 'value 1';
   public $var2 = 'value 2';
   public $var3 = 'value 3';

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

}

$class = new MyClass();

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

Результат:

|  |
| --- |
| var1 => value 1var2 => value 2var3 => value 3 |

Как показывает результат, **foreach** проитерировал все принадлежащие объекту **public**-свойства.

**PHP5 и ООП (Часть 3)**

1. **\_\_autoload()**

Многие разработчики, пишущие объектно-ориентированные приложения, создают один файл, в котором содержится определение класса. Очень неудобно писать в начале каждого скрипта длинный список включаемых файлов по одному на каждый класс.

В PHP 5 в этом больше нет необходимости. Вы можете определить функцию **\_\_autoload()** , которая автоматически будет вызываться в случае использования класса, который не был определен выше. Вызывая такую функцию, Zend Engine дает возможность загрузить файл с определением класса прежде, чем будет сформировано сообщение об ошибке и выполнение скрипта прекратится.

<?php
function \_\_autoload($class\_name) {
     include\_once($class\_name . "php");
}

$obj  = new MyClass1();
$obj2 = new MyClass2();
?>

1. **Обработка исключительных ситуаций (исключений)**

PHP 5 добавляет парадигму обработки исключений, вводя структуру **try/throw/catch**. Вам остается только создать объекты, которые наследуют класс исключений Exception.

<?php
class SQLException extends Exception {
     public $problem;
     function \_\_construct($problem) {
         $this->problem = $problem;
     }
}

try {
     ...
     throw new SQLException("Couldn't connect to database");
     ...
} catch (SQLException $e) {
     print "Caught an SQLException with problem $obj->problem";
} catch (Exception $e) {
     print "Caught unrecognized exception";
}
?>

В настоящее время в целях обратной совместимости большинство внутренних функций не использует исключения. Однако, все новые расширения будут иметь такую возможность, и вы можете использовать такую конструкцию в своем исходном тексте. Кроме того, подобно уже существующей функции [**set\_error\_handler()**](http://www.php.su/functions/?set_error_handler), вы можете использовать [**set\_exception\_handler()**](http://www.php.su/functions/?set_exception_handler), чтобы отловить необработанное исключение прежде, чем выполнение скрипта будет закончено.

1. **Сравнение объектов**

В PHP 5 сравнение объектов является более сложным процессом, чем в PHP 4, а также процессом, более соответствующим идеологии объектно-ориентированного языка.

При использовании оператора сравнения (*==*), свойства объектов просто сравниваются друг с другом, а именно: два объекта равны, если они содержат одинаковые свойства и одинаковые их значения и являются экземплярами одного и того же класса.

С другой стороны, при использовании оператора идентичности (*===*), свойства объекта считаются идентичными тогда и только тогда, когда они ссылаются на один и тот же экземпляр одного и того же класса.

Следующий пример внесёт ясность.

<?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 {
     var $flag;

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

class OtherFlag {
     var $flag;

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

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

echo "Два экземпляра одного и того же класса\n";
compareObjects($o, $p);

echo "\nДве ссылки на один и тот же экземпляр\n";
compareObjects($o, $q);

echo "\nЭкземпляры двух разных классов\n";
compareObjects($o, $r);
?>

Результатом выполнения этого кода будет:

Два экземпляра одного и того же класса
o1 == o2 : TRUE
o1 != o2 : FALSE
o1 === o2 : FALSE
o1 !== o2 : TRUE

Две ссылки на один и тот же экземпляр
o1 == o2 : TRUE
o1 != o2 : FALSE
o1 === o2 : TRUE
o1 !== o2 : FALSE

Экземпляры двух разных классов
o1 == o2 : FALSE
o1 != o2 : TRUE
o1 === o2 : FALSE
o1 !== o2 : TRUE

**PHP5 и ООП (Часть 4)**

1. **Перегрузка свойств классов**

Обращения к свойствам объекта могут быть перегружены с использованием методов **\_\_call**, **\_\_get** и **\_\_set**. Эти методы будут срабатывать только в том случае, если объект или наследуемый объект не содержат свойства, к которому осуществляется доступ. Синтаксис такой:

void **\_\_set** ( string имя, mixed значение )

void **\_\_get** ( mixed имя )

С помощью этих методов обращения к свойствам класса могут быть перегружены с целью выполнения произвольного кода, описанного в классе. В аргументе *имя* передаётся имя свойства, к которому производится обращение. Аргумент *значение* метода \_\_set() должен содержать значение, которое будет присвоено свойству класса с именем *имя*.

Пример перегрузки с использованием **\_\_get** и **\_\_set**:

<?php
class Setter {
   public $n;
   private $x = array("a" => 1, "b" => 2, "c" => 3);

   function \_\_get($nm) {
     print "Читаем [$nm]\n";

     if (isset($this->x[$nm])) {
       $r = $this->x[$nm];
       print "Получили: $r\n";
       return $r;
     } else {
       print "Ничего!\n";
     }
   }

   function \_\_set($nm, $val) {
     print "Пишем $val в [$nm]\n";

     if (isset($this->x[$nm])) {
       $this->x[$nm] = $val;
       print "OK!\n";
     } else {
       print "Всё плохо!\n";
     }
   }
}

$foo = new Setter();
$foo->n = 1;
$foo->a = 100;
$foo->a++;
$foo->z++;
var\_dump($foo);
?>

Результатом выполнения рассмотренного скрипта будет:

Пишем 100 в [a]
OK!
Читаем [a]
Получили: 100
Пишем 101 в [a]
OK!
Читаем [z]
Ничего!
Пишем 1 в [z]
Всё плохо!
object(Setter)#1 (2) {
   ["n"]=>
   int(1)
   ["x:private"]=>
   array(3) {
     ["a"]=>
     int(101)
     ["b"]=>
     int(2)
     ["c"]=>
     int(3)
   }
}

1. **Перегрузка методов**

Вызовы методов могут быть перегружены с использованием методов **\_\_call**, **\_\_get** и **\_\_set**. Эти методы будут срабатывать только в том случае, если объект или наследуемый объект не содержат метода, к которому осуществляется доступ. Синтаксис:

mixed **\_\_call** ( string имя, array аргументы )

С использованием этого метода, методы класса могут быть перегружены с целью выполнения произвольного кода, описанного в классе. В аргументе *имя* передаётся имя вызванного метода. Аргументы, которые были переданы методу при обращении, будут возвращены чере *аргументы*. Значение, возвращённое методом**\_\_call()**, будет передано вызывающему оператору.

Пример перегрузки с использованием **\_\_call**:

<?php
class Caller {
   private $x = array(1, 2, 3);

   function \_\_call($m, $a) {
     print "Вызван метод $m :\n";
     var\_dump($a);
     return $this->x;
   }
}

$foo = new Caller();
$a = $foo->test(1, "2", 3.4, true);
var\_dump($a);
?>

Результат выполнения рассмотренного примера:

Вызван метод test:
array(4) {
   [0]=>
   int(1)
   [1]=>
   string(1) "2"
   [2]=>
   float(3.4)
   [3]=>
   bool(true)
}
array(3) {
   [0]=>
   int(1)
   [1]=>
   int(2)
   [2]=>
   int(3)
}

1. **Интерфейсы**

Интерфейсы объектов позволяют программисту создавать код, который указывает, какие методы и свойства должен включать класс, без необходимости описывания их функционала.

Интерфейсы объявляются так же, как и обычные классы, но с использованием ключевого слова "**interface**"; тела методов интерфейсов должны быть пустыми. Для включения интерфейса в класс программист должен использовать ключевое слово "**implements**" и описать функционал методов, перечисленных во включаемом интерфейсе. Если это требуется, классы могут включать более одного интерфейса путём их перечисления через пробел.

Если класс включает какой-либо интерфейс и не описывает функционал всех методов этого интерфейса, выполнение кода с использованием такого класса завершится фатальной ошибкой, сообщающей, какие именно методы не были описаны. Пример интерфейса:

<?php
interface ITemplate
{
   public function setVariable($name, $var);
   public function getHtml($template);
}

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;
   }
}
?>

1. **Оператор instanceof**

Поддержка проверки зависимости от других объектов. Функцией is\_a(), известной из PHP 4, пользоваться теперь не рекомендуется.

<?php
if ($obj instance of Circle) {
     print '$obj is a Circle';
}
?>

1. **Метод final**

Ключевое слово final позволяет вам помечать методы, чтобы наследующий класс не мог перегрузить их. Разместив перед объявлениями методов или свойств класса ключевое слово **final**, вы можете предотвратить их переопределение в дочерних классах, например:

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

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

class ChildClass extends BaseClass {
    public function moreTesting() {
        echo "Вызван метод ChildClass::moreTesting()\n";
    }
}
// Выполнение заканчивается фатальной ошибкой:
//Cannot override final method BaseClass::moreTesting()
// (Метод BaseClass::moretesting() не может быть переопределён)
?>

1. **Классы, помеченные как final**

После объявления класса **final** он не может быть унаследован. Следующий пример вызовет ошибку:

<?php
final class FinalClass {
}

class BogusClass extends FinalClass {
}
?>

Более подробно о возможностях PHP5 и Zend 2.0 вы можете узнать, обратившись к документации [Zend Engine 2.0](http://www.zend.com/engine2/zendengine-2.0.pdf).

## Функции для работы с классами и объектами

В РНР существует несколько стандартных функций для работы с классами и объектами. Рассмотрим некоторые функции для работы с классами и объектами в контексте PHP4.

**get\_class\_methods()**

Функция [**get\_class\_methods()**](http://www.php.su/functions/?get_class_methods) возвращает массив имен методов класса с заданным именем. Синтаксис функцииget\_class\_methods():

array get\_class\_methods (string имя\_класса)

Простой пример использования get\_class\_methods() - Получение списка методов класса:

<?php
...
class Airplane extends Vehicle {
  var $wingspan;
  function setWingSpan($wingspan) {
    $this->wingspan = $wingspan;
  }

  function getWingSpan() {
    return $this->wingspan;
  }
}

$cls\_methods = get\_class\_methods(Airplane);
// Массив $cls\_methods содержит имена всех методов,
// объявленных в классах "Airplane" и "Vehicle"
?>

Как видно из листинга, функция get\_class\_methods() позволяет легко получить информацию обо всех методах, поддерживаемых классом.

**get\_class\_vars()**

Функция [**get\_class\_vars()**](http://www.php.su/functions/?get_class_vars) возвращает массив имен атрибутов класса с заданным именем. Синтаксис функцииget\_class\_vars():

array get\_class\_vars (string имя\_класса)

Пример использования функции get\_class\_vars() - получение списка атрибутов (свойств) класса:

<?php

class Vehicle {
var $model;
var $current\_speed;
}

 class Airplane extends Vehicle {
 var $wingspan;
 }

$a\_class = "Airplane";
$attribs = get\_class\_vars($a\_class);
// $attribs = array ("wingspan", "model", "current\_speed")
?>

В рассмотренном примере массив $attribs заполняется именами всех атрибутов класса Airplane.

**get\_object\_vars()**

Функция [**get\_object\_vars()**](http://www.php.su/functions/?get_object_vars) возвращает ассоциативный массив с информацией обо всех атрибутах объекта с заданным именем. Синтаксис функции get\_object\_vars():

array get\_object\_vars (object имя\_обьекта)

Пример использования функции get\_object\_vars() - получение информации о переменных объекта:

<?php

class Vehicle {
var $wheels;
}

 class Land extends Vehicle {
 var $engine;
 }

  class car extends Land {
  var $doors;
   function car($doors, $eng, $wheels) {
   $this->doors = $doors;
   $this->engine = $eng;
   $this->wheels = $wheels;
   }

   function get\_wheels() {
   return $this->wheels;
   }
  }

$toyota = new car(2,400,4);
$vars = get\_object\_vars($toyota);
while (list($key, $value) = each($vars)) :
print "$key ==> $value <br>";
endwhile;
// Выходные данные:
// wheels ==> 4
// engine ==> 400
// doors ==> 2

?>

Функция get\_object\_vars() позволяет быстро получить всю информацию об атрибутах конкретного объекта и их значениях в виде ассоциативного массива.

**method\_exists()**

Функция [**method\_exists()**](http://www.php.su/functions/?method_exists) проверяет, поддерживается ли объектом метод с заданным именем. Если метод поддерживается, функция возвращает **TRUE**, в противном случае возвращается **FALSE**. Синтаксис функции method\_exists():

bool method\_exi sts (object имя\_обьекта. string имя\_метода)

Пример использования метода method\_exists() - проверка поддержки метода объектом:

<?php

class Vehicle {
// ...
}

 class Land extends Vehicle {
 var $fourWheel;

  function setFourWheelDrive() {
  $this->fourWeel = 1;
  }

 }
// Создать объект с именем $саr
$car = new Land;
// Если метод "fourWheelDrive" поддерживается классом "Land"
// или "Vehicle", вызов method\_exists возвращает TRUE;
// в противном случае возвращается FALSE.
// В данном примере method\_exists() возвращает TRUE.
if (method\_exists($car, "setfourWheelDrive")) :
print "This car is equipped with 4-wheel drive";
else :
print "This car is not equipped with 4-wheel drive";
endif;
?>

В рассмотренном примере функция method\_exists() проверяет, поддерживается ли объектом $car метод с именемsetFourWheelDrive(). Если метод поддерживается, функция возвращает логическую **истину** и фрагмент выводит соответствующее сообщение. В противном случае возвращается **FALSE** и выводится другое сообщение.

**get\_class()**

Функция [**get\_class()**](http://www.php.su/functions/?get_class) возвращает имя класса, к которому относится объект с заданным именем. Синтаксис функцииget\_class():

string get\_class(object имя\_объекта);

Пример использования get\_class() - получение имени класса:

<?php

class Vehicle {
}

 class Land extends Vehicle {
 }

// Создаем объект с именем $саr:
$car = new Land;
// Переменной $class\_a присваивается строка "Land":
$class\_a = get\_class($car);
echo $class\_a;
?>

В рассмотренном примере переменной **$class\_a** присваивается имя класса, на основе которого был создан объект **$саr**.

**get\_parent\_class()**

Функция [**get\_parent\_class()**](http://www.php.su/functions/?get_parent_class) возвращает имя родительского класса (если он есть) для объекта с заданным именем. Синтаксис функции get\_parent\_dass():

string get\_parent\_class (object имя\_обьекта);

Пример получения имени родительского класса функцией get\_parent\_class():

<?php

class Vehicle {
//...
}

 class Land extends Vehicle {
 //...
 }

// Создаем объект с именем $саr:
$саr = new Land;
// Переменной $parent присваивается строка "Vehicle":
$parent = get\_parent\_class($car);
?>

При вызове get\_parent\_class() переменной $parent будет присвоена строка "Vehicle".

**is\_subclass\_of()**

Функция [**is\_subclass\_of()**](http://www.php.su/functions/?is_subclass_of) проверяет, был ли объект создан на базе класса, имеющего родительский класс с заданным именем. Функция возвращает **TRUE**, если проверка дает положительный результат, и **FALSE** в противном случае. Синтаксис функции is\_subclass\_of():

bool is\_subclass\_of (object объект, string имя\_класса)

Пример использования функции is\_subdass\_of():

<?php

class Vehicle {
//...
}

 class Land extends Vehicle {
 //...
 }
$auto = new Land;
// Переменной $is\_subclass присваивается TRUE
$is\_subclass = is\_subclass\_of($auto, "Vehicle");
?>

В рассмотренном примере переменной $is\_subclass() присваивается признак того, принадлежит ли объект $auto к субклассу родительского класса Vehicle. В приведенном фрагменте $auto относится к классу Vehicle; следовательно $is\_subclass()будет присвоено значение **TRUE**.

**get\_declared\_classes()**

Функция [**get\_declared\_classes()**](http://www.php.su/functions/?get_declared_classes) возвращает массив с именами всех определенных классов. Синтаксис функцииget\_declared\_classes():

array get\_declared\_classes()

Пример получения списка классов функцией get\_declared\_classes():

<?php

 class Vehicle {
 //...
 }

  class Land extends Vehicle {
  //...
  }

$declared\_classes = get\_declared\_classes();
// $declared\_classes = array("Vehicle", "Land")
?>

Мы рассмотрели лишь некоторые основные функции, предназначенные для работы с классами и объектами PHP. Для ознакомления с полным перечнем таких функций обратитесь к справочнику функций PHP.