РАЗРАБОТКА WEB-ПРИЛОЖЕНИЙ НА PHP&MYSQL

УРОК 5

ООП в PHP (Часть 2)

ВВЕДЕНИЕ

В этом уроке мы продолжим знакомство с реализацией концепции **ООП** в **PHP5** и сосредоточимся на более продвинутых средствах и решениях, помогающих облегчить использование **ООП** при создании современных web-приложений.

В частности, мы рассмотрим реализацию полиморфизма в **PHP5**. Особенно с учётом того, что, по сравнению с **PHP4**, она сильно доработана. Разберем специальные методы, создающиеся в классе во время выполнения сценария. Их можно перегружать в любом классе. А также поговорим о стандартных функциях **PHP** для работы с классами.

1 ПОЛИМОРФИЗМ

1.1 Абстрактные классы и методы

Основной целью, которая преследовалась при создании **PHP 5**, было достижение максимального соответствия концепциям **ООП**. Поэтому в **PHP 5** также появилась возможность создания абстрактных методов и классов.

Чтобы понять, что определяет абстрактный объект, давайте рассмотрим такое понятие, как “пища”. Все мы знаем, что такое “пища”, но не всегда смотрим на то, из чего конкретно она приготовлена. То есть само по себе понятие пищи является абстрактным — оно существует только как обобщение более конкретных вещей.

Подобная идея справедлива также и для абстрактных классов.

В **ООП** абстрактные классы предназначены для того, чтобы создать суперкласс, который будет определять абстрактные характеристики его классов-наследников. На самом деле абстрактные классы могут содержать в себе какой-либо код, а могут быть вообще без кода; кроме этого, на их основе нельзя создать экземпляр напрямую. Рассмотрим пример:

ПРИМЕР 1.1.1

```
<?php
abstract class Number {
    private $value;
    abstract public function value();
```

```
    public function reset() {
        $this->value=NULL;
    }
}

class Integer extends Number {
    private $value;
    public function value() {
        return (int)$this->value;
    }
}

$num = new Integer; /* Правильно */
$num2 = new Number; /* Не правильно */
?>
```

Создан абстрактный класс `Number`, который расширяет класс `Integer`. Поскольку класс `Number` объявлен как `abstract`, на его основе нельзя строить экземпляры. Можно увидеть, что в классе `Number` определены две функции: `value()` и `reset()`. Абстрактный класс может не содержать код методов, хотя при необходимости его можно добавить.

Что же касается класса `Number`, то поскольку функция `value()` является специфической для конкретного типа числа, она реализуется в классе-наследнике. Чтобы разработчик мог реализовать такое поведение в своем коде, используется ключевое слово `abstract`, указывающее на то, что это просто заполнитель в классе `Number`. Однако это не относится к методу `reset()`, который остается неизменным для любого конкретного типа числа.

> Абстрактные классы можно расширять, не реализовывая все абстрактные методы, определенные в них. Другими словами, класс-наследник также должен быть определен как абстрактный класс, если он не реализует все абстрактные методы своего родительского класса.

Рассмотрим еще один пример, посвященный полиморфизму.

ПРИМЕР 1.1.2

```
<?php
class Figure() {
    function DrawSelf() { echo "Рисуем абстрактную фигуру";
    function Draw() { $this->DrawSelf(); }
}

class Rectangle extends Figure {
    function DrawSelf() { echo "Рисуем прямоугольник";
```

```
}

$a=new Figure();
$b=new Rectangle();
$a->Draw();
$b->DrawSelf();
$b->Draw();

/*
Вывод:

Рисуем абстрактную фигуру
Рисуем прямоугольник
Рисуем прямоугольник



*/
?>
```

Предположим, что класс `Figure` определяет какую-нибудь абстрактную геометрическую фигуру – мы еще не знаем что это за фигура – квадрат, эллипс или треугольник Функция `DrawSelf()` рисует эту фигуру. Конечно, рисовать мы ничего не станем, а просто выведем на экран наши намерения что-нибудь нарисовать. Функция `Draw()` — это просто ссылка на функцию `DrawSelf()`.

Производный класс `Rectangle` также обладает функцией `DrawSelf()`, но эта функция рисует не произвольную фигуру, а прямоугольник. Эта функция не имеет параметров, но, как мы уже знаем, мы можем указывать произвольное число параметров для функции, не определяя их явно.

Как вы видите, при вызове функции `Draw()` класса `Rectangle()` будет нарисован прямоугольник, а не неопределенная фигура. Вот это и есть проявление полиморфизма: функция `Draw()` класса `Figure()` была перезаписана одноименной функцией класса `Rectangle()`.

Функция, переопределенная таким образом, называется **виртуальной**.

1.2 Интерфейсы

В отличие от абстрактных классов, которыми можно выражать некие абстрактные понятия, т.е. не до конца детализированные на том или ином уровне абстракции, интерфейсы нужны для того, что-

бы обеспечить определенную функциональность внутри класса. Если выражаться точнее, то интерфейс представляет собой средство для определения набора методов, которые обязан содержать некий класс, реализующий данный интерфейс.

Использование интерфейсов — это один из вариантов обеспечения полиморфизма в ООП. Все классы, реализующие один и тот же интерфейс, ведут себя идентично, с точки зрения той функциональности, которая реализуется этим интерфейсом. Это позволяет оперировать общими методами обработки объектов разных типов в программах, но получать один и тот же результат при этом.

Интерфейс, при создании, объявляется ключевым словом `interface`. В остальном, синтаксис похож на описание обычного класса. Ниже рассмотрен пример интерфейса для вывода на страницу некоторого содержимого объекта.

ПРИМЕР 1.2.1

```
<?php
interface printable {
    public function printme();
}
?>
```

Чтобы интерфейс приносил пользу, он должен быть реализован с помощью одного или нескольких классов. В примере определен интерфейс `printable`, который заявляет, что любой класс, реализующий этот интерфейс, должен реализовать метод `printme()`. Чтобы создать класс, реализующий подобный интерфейс, в определении класса используется ключевое слово `implements`, за которым следует список реализованных интерфейсов.

ПРИМЕР 1.2.2

```
<?php
class Integer implements printable {
    private $value;

    public function getValue() {
        return (int)$this->value;
    }

    public function printme() {
        echo (int)$this->value;
    }
}
?>
```

Здесь определен исходный класс `Integer`, чтобы реализовать

интерфейс printable. Как класс, реализующий этот интерфейс (см. ключевое слово implements), он гарантирует, что класс Integer предложит все методы, которые определены в интерфейсе.

Теперь, когда определено, что класс реализует заданный интерфейс, можно гарантировать, что любые функции или методы, для которых требуются некоторые действия со стороны класса, получат их. Эти способы можно использовать для определения интерфейса printable.

ПРИМЕР 1.2.3

```
<?php
interface printable {
    public function printme();
}

abstract class Number {
    private $value;
    abstract public function value();
    public function reset() {
        $this->value = NULL;
    }
}

class Integer extends Number implements printable {
    private $value;
    function __construct($value) {
        $this->value = $value;
    }
    public function getValue() {
        return (int)$this->value;
    }
    public function printme() {
        echo (int)$this->value;
    }
}

/* Создание функции, которую требует интерфейс printable. */
function printNumber(printable $myObject) {
/* Если эта функция будет вызвана, мы точно знаем, */
/*что она имеет метод printme() */
    $myObject->printme();
}
$inst = new Integer(10);
printNumber($inst);
?>
```

Здесь интерфейсы использовались для гарантии того, что функ-

ция `printNumber()` будет всегда получать объект, имеющий метод `printme()`. Другая полезная особенность заключается в том, что один класс может реализовать несколько различных интерфейсов.

ПРИМЕР 1.2.4

```
<?php
interface printable {
    public function printme();
}
interface Inumber {
    public function reset();
}

class Integer implements printable, Inumber {
    private $value;
    function __construct($value) {
        $this->value = $value;
    }
    public function printme() {
        echo (int)$this->value;
    }
    public function reset() {
        $this->value = NULL;
    }
    public function value() {
        return (int)$this->value;
    }
}

function resetNumber(Inumber $obj) {
    $obj->reset();
}

function printNumber(printable $obj) {
    $obj->printme();
}

$inst = new Integer(10);
printNumber($inst);
resetNumber($inst);
?>
```

Следует заметить, что класс может быть наследником одного класса и при этом обязательно поддерживать несколько интерфейсов. При этом указание базового класса должно обязательно следовать до указания перечня реализуемых методов.

1.3 Практический пример: реализация интерфейсов для класса

Давайте вернёмся к примеру электронного магазина, который мы начали реализовывать в прошлом уроке (см. урок № **4**, раздел **2.5**, стр. **9**).

Напомню, что в прошлом уроке мы создали два файла:

- `commodity.php` – файл, который содержит описание класса товара;
- `book.php` – файл, который содержит описание класса более конкретного товара – книги, который наследует класс товара из предыдущего файла.

Создадим два интерфейса: один для определения методов работы с базой данных (работа с базами данных будет рассмотрена в уроках **8** и **9**, поэтому сами методы мы, конечно же, пока реализовывать не будем – подождём…, т.е. сделаем задел на будущее. ☺), а другой для определения методов работы с подсистемой статистики.

Создадим файл `interfaces.php` с описанием необходимых нам интерфейсов.

ПРИМЕР 1.3.1

```
<?php
interface Database {
    function select(); /* Метод для получения выборки данных из таблицы базы данных */
    function insert(); /* Метод для записи данных в базу */
    function update(); /* Метод для обновления данных в таблице */
    function delete(); /* Метод для удаления данных из таблицы */
}
interface Stats {
    function increaseViews();
    function increaseOrders();
}
?>
```

Подключим его к файлу с описанием класса книги (`book.php`) и реализуем в нём эти интерфейсы.

ПРИМЕР 1.3.2

```
<?php
require_once 'commodity.php'; /* Мы подключаем в качестве библиотеки файл, в котором описан класс товара. */

require_once 'interfaces.php'; /* Мы подключаем в качестве
```

```
библиотеки файл, в котором описаны интерфейсы */

class Book extends Commodity implements Database, Stats {
    public $authors;
    function __construct($name, $authors,
                         $category, $price=null,
                         $availability=false) {
        parent::__construct($name, $category, $price,
                         $availability);
        $this->authors=$authors;
    }
    function getAuthors() {
        return $this->authors;
    }
    function select() {
        // Тело метода select()
    }
    function insert() {
        // Тело метода insert()
    }
    function update() {
        // Тело метода update()
    }
    function delete() {
        // Тело метода delete()
    }
    function increaseViews() {
        // Тело метода increaseViews()
    }
    function increaseOrders() {
        // Тело метода increaseOrders()
    }
}
?>
```

2 СПЕЦИАЛЬНЫЕ МЕТОДЫ

Среди средств работы с классами и объектами, имеющимися в арсенале разработчика **PHP**, есть некоторые специальные методы, которые можно использовать в классах. Хотя вы уже знакомы с некоторыми специальными методами, такими как `__construct()`, `__destruct()` и `__clone()`, но существуют еще и другие аналогичные методы.

Всех их объединяет то, что они вызываются интерпретатором

самостоятельно, автоматически тогда, когда возникает соответствующая ситуация в программе.

2.1 Метод-получатель и метод-установщик

Метод-получатель `__get()` и метод-установщик `__set()` используются как универсальный интерфейс при обращении к свойствам в объектах. Эти методы вызываются в том случае, если свойство не было определено ранее. Вот прототипы этих специальных методов:

Пример 2.1.1

```
function __get($name);
function __set($name, $value);
```

`$name` – это имя переменной, к которой обращается сценарий, но которой не существует. Несложно предположить, аргумент `$value` метода `__set()` соответствует новому значению, которое будет присвоено вместо несуществующего значения.

Метод-получатель и метод-установщик вызываются только в том случае, если требуемого свойства вообще нет в объекте. Если данное свойство изначально не существовало, но в какой-то момент было добавлено в экземпляр посредством метода-установщика:

Пример 2.1.2

```
<?php
function __set($name, $value) {
    $this->$name = $value;
}
?>
```

то в будущем ни метод `__get()`, ни метод `__set()` не будут вызываться.

Метод-получатель и метод-установщик полезно использовать, например, при работе с Web-службами или контейнерными объектами, в которых свойства, доступные в экземпляре класса, не известны до тех пор, пока не будет начато выполнение сценария.

Например, если бы мы решили хранить все свойства товаров нашего интернет-магазина в массиве, то мы могли бы использовать эти методы:

Пример 2.1.3

```
<?php
class Commodity {
    private $properties;
    function __set($name, $value) {
```

```
        echo "задание нового свойства $name = $value";
        $this->$properties[$name] = $value;
    }
    function __get($name, $value) {
        echo "чтение значения свойства $name";
        return $this->$properties[$name];
    }
}
?>
```

При этом работа с товаром выглядела бы так.

ПРИМЕР 2.1.4

```
<?php
$book=new Book('Разработка web-приложений на PHP5', 'Программирование');
$book->weight=100;
// Выведет 'задание нового свойства weight=100'
$weight=$book->weight;
// Выведет 'чтение значения свойства weight'
echo $weight;
?>
```

2.2 Метод __call()

Подобно методу-получателю и методу-установщику, которые позволяют динамически обрабатывать доступ к свойствам в **PHP**-сценариях, метод `__call()` служит для того, чтобы организовать хранилище вызовов методов в объекте. При получении вызова метода, который не был определен в классе, по возможности вызывается метод `__call()`.

Прототип этого метода такой:

ПРИМЕР 2.2.1

```
function __call($method, $arguments);
```

где `$method` — это строка, соответствующая вызванному методу, а `$arguments` — массив, содержащий параметры, передаваемые этому методу.

Подобно методу-получателю и методу-установщику, метод `__call()` полезно использовать в тех случаях, когда весь список функций не доступен вплоть до начала выполнения сценария. Как вариант, метод `__call()` можно использовать для того, чтобы создать универсальный метод для обработки вызовов недействительных методов в **PHP**-сценариях.

ПРИМЕР 2.2.2

```
<?php
class Commodity {
    function __call($method, $params) {
        echo "Метод $method не существует!\n";
        echo "Переданные параметры: <pre>";
        print_r($params);
        echo "</pre>";
    }
}
$book=new Book('Разработка web-приложений на PHP5', 'Про-
граммирование');
$book->nonExistentFunction(1, 'test');
?>
```

Во время выполнения этого кода будет вызван метод, который до этого не был определен. Однако вместо того чтобы генерировать неустранимую ошибку, недействительный вызов инициирует вызов метода `__call()`, в результате чего у вас будет возможность исправить ошибку.

2.3 Метод __toString()

Предназначен для упрощения строкового представления сложного объекта. Если определить этот метод, **PHP** будет вызывать его в тех случаях, когда объект лучше всего обрабатывать как строку (чаще всего при отображении с использованием операторов `echo` или `print`). Перегрузив этот метод, мы укажем то, как будет выглядеть объект в строковом представлении.

ПРИМЕР 2.3.1

```
<?php
class Book extends Commodity {
    public $autors;
    //…
    function __toString() {
        foreach ($this->authors as $author) {
            echo $author.", ";
        }
        echo $this->name;
        echo ($availability ? '(Есть на складе)' : '(нет на
складе)');
    }
}
?>
```

2.4 Автоматическая загрузка классов

В **PHP 4** при создании классов у разработчиков не было доступного механизма автоматической загрузки определенного класса по требованию. Большинство разработчиков пишут свои приложения, размещая каждый класс в отдельном файле для удобства управления ими. Но такой подход вынуждает всякий раз в начале файла перечислять подключаемые файлы, а такой список может быть достаточно большим.

В **PHP 5** все иначе: можно определить функцию для загрузки классов по мере необходимости. Это функция `__autoload()`; она имеет следующий прототип:

Пример 2.4.1

```
function __autoload($classname);
```

где `$classname` — это имя класса, который мы желаем сделать найденным **PHP**.

В функцию `__autoload()` можно добавить любую логику для определения местоположения класса и его загрузки. Благодаря этому можно быстро и легко осуществить удаленную загрузку классов из базы данных, файловой системы и тому подобного. Единственное, что необходимо сделать в случае использования этой функции, это загрузить класс в **PHP** (обычно с помощью оператора `require_once()`) до окончания вызова функции. Если класс не был загружен до завершения выполнения функции `__autoload()` (или если функция `__autoload()` не была определена), **PHP** завершит выполнение сценария и выведет сообщение о невозможности найти определенный класс.

Пример 2.4.2

```
<?php
function __autoload($class) {
    $files = array('MyClass' => "/path/to/myClass.class.php",
    'anotherClass' => "/path/to/anotherClass.class.php");
    if(!isset($files[$class])) return;
    require_once($files[$class]);
}
$a = new MyClass;
$b = new anotherClass;
?>
```

Функция `__autoload()` используется для загрузки классов, определяя их местоположение в заранее заданном ассоциативном массиве. Поскольку классы `MyClass` и `anotherClass` еще не были

определены в сценарии, в обоих случаях будет вызываться функция __autoload(), для того чтобы сценарий самостоятельно смог найти требуемый класс.

2.5 Методы __sleep() и __wakeup()

Существуют еще два специальных метода: __sleep() и __wakeup(). Они применяются для определения поведения объекта при его упаковке и, соответственно, распаковке с помощью функций serialize() и unserialize().

Для начала, нужно рассказать о **сериализации**, т.е. о функции serialize(). Эта функция предназначена для генерации строки, являющейся пригодным для хранения представлением сложной переменной, например массива или объекта какого-либо класса.

```
string serialize (mixed value)
```

Например:

Пример 2.5.1

```
<?php
$A=array("a"=>"aa", "b"=>"bb", "c"=>array("x"=>"xx"));
$st=serialize($A);
echo $st;
/* выведется следующее:
a:2:{s:1:"a";s:2:"aa";s:1:"b";s:2:"bb";s:1:"c";a:1:{s:1:"x";
s:2:"xx";}}
*/
?>
```

Функция unserialize(), наоборот, принимает в качестве параметра $st строку, ранее созданную при помощи serialize(), и возвращает целиком объект, который был упакован.

```
mixed unserialize(string st)
```

Например:

Пример 2.5.2

```
<?php
$a=array(1,2,3);
$s=serialize($a);
$a="bogus";
echo count($a); // выводит 1
$a=unserialize($s);
echo count($a); // выводит 3
?>
```

Еще раз отмечу: сериализовать можно не только массивы, но и вообще что угодно. Механизм сериализации часто применяется также и для того, чтобы сохранить какой-то объект в базе данных, и тогда без сериализации практически не обойтись. Но о базах данных и работе с ними позже.

Итак, применение функций сериализации к объектам.

Функция `serialize()` проверяет, присутствует ли в вашем классе метод с "магическим" именем `__sleep()`. Если это так, то этот метод выполняется прежде любой операции сериализации. Он должен возвращать индексированный массив строк, представляющих имена свойств, которые необходимо включить в создаваемую последовательность. Любые свойства, не указанные в этом списке, не будут сохранены в последовательности.

Обычно `__sleep()` используется для передачи ожидаемых данных или для выполнения обычных задач их очистки. Также, этот метод можно выполнять в тех случаях, когда вы не хотите сохранять очень большие объекты полностью.

С другой стороны, функция `unserialize()` проверяет наличие метода с "магическим" именем `__wakeup()`. Если такой имеется, то он может воссоздать все ресурсы объекта, которые тот имеет.

Обычно `__wakeup()` используется для восстановления любых соединений с базой данных, которые могли быть потеряны во время операции сериализации и выполнения других операций повторной инициализации.

ПРИМЕР 2.5.3

```
<?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();
        }
}
?>
```

В этом примере мы совсем немножко забегаем вперед, применяя функции `mysql_connect()` и `mysql_select_db()`. Эти функции имеют вполне понятные названия и осуществляют соединение с сервером баз данных (`$server`) и, затем, выбор нужной на этом сервере базы (`$db`). Для соединения с сервером используется логин (`$username`) и пароль (`$password`).

3 ОПЕРАТОР ПРОВЕРКИ ПРИНАДЛЕЖНОСТИ К КЛАССУ

Оператор `instanceof` используется для определения того, явля-

ется ли текущий объект экземпляром указанного класса.

Оператор `instanceof` был добавлен в **PHP 5**. До этого использовалась конструкция `is_a()`, которая на данный момент не рекомендуется к применению, более предпочтительно использовать оператор `instanceof`.

ПРИМЕР 3.1.1

```
<?php
class A { }
class B { }
$thing = new A;
if ($thing instanceof A) {
    echo 'A';
}
if ($thing instanceof B) {
    echo 'B';
}
?>
```

Поскольку объект `$thing` является экземпляром класса `A`, и никак не `B`, то будет выполнен только первый, опирающийся на класс `A`, блок, т.е. выведено будет `A`.

4 ФУНКЦИИ ДЛЯ РАБОТЫ С КЛАССАМИ

Мы завершаем рассмотрение концепции ООП и её реализации в PHP 5.

В конце мы рассмотрим несколько функций ядра, направленных на работу с классами. Они позволят получить информацию о классах и экземплярах объектов: имя класса из экземпляра объекта, равно как и все его свойства и методы, можно установить не только принадлежность объекта к конкретному классу, но и определить порядок наследования (к примеру, какой класс наследует класс данного объекта).

Итак, функции.

```
bool class_exists(string class_name)
```

Функция возвращает `true` если класс **`class_name`** был объявлен. В противном случае функция возвращает `false`.

```
string get_class(object obj)
```

Функция возвращает имя класса, экземпляром которого является объект **`obj`**. Если **`obj`** не является объектом, функция вернет

false.

ПРИМЕР 4.1.1

```
<?php
class foo {
    function __construct() { return (true); }
    function name() { echo "Моё имя ".get_class($this)."\n";}
}
$bar=new foo(); // создание объекта
echo "Имя класса ".get_class($bar)."\n"; // внешний вызов
$bar->name(); // внутренний вызов
?>
/* выведет:
Имя класса foo
Моё имя foo
*/
```

array **get_class_methods**(mixed **class_name**)

Функция возвращает массив имен методов определенных для класса **class_name**.

Пример использования get_class_methods()

ПРИМЕР 4.1.2

```
<?php
class myclass {
    function __construct() { return(true); }
    function myfunc1() { return(true); }
    function myfunc2() { return(true); }
}


$my_object = new myclass();
$class_methods = get_class_methods(get_class($my_object));
foreach ($class_methods as $method_name) {
    echo "$method_name ";
}
?>
// выведет: __construct myfunc1 myfunc2
```

array **get_class_vars**(string **class_name**)

Как несложно догадаться, функция возвращает ассоциативный массив свойств класса и их значения по-умолчанию.

ПРИМЕР 4.1.3

```
<?php
class myclass {
    public $var1; // переменная без начального значения...
```

```
    public $var2="xyz";
    public $var3=100;
    function __construct() {
        $this->var1="новое значение";
        $this->var2="и тут тоже";
        return true;
    }
}
$my_class=new myclass();
$class_vars=get_class_vars(get_class($my_class));
foreach ($class_vars as $name=>$value) {
    echo "$name : $value\n";
}
?>
/* выведет:
var1 :
var2 : xyz
var3 : 100
*/
```

Если же необходимо получить список свойств, имеющихся непосредственно в определенном экземпляре объекта, а также реальные значения, присвоенные им, то можно воспользоваться другой функцией:

```
array get_object_vars(object obj)
```

Функция возвращает ассоциативный массив объявленных свойств класса и их текущих значений для объекта **obj**.

ПРИМЕР 4.1.4

```
<?php
class Point2D {
    public $x, $y;
    public $label;
    function __construct($x, $y) {
        $this->x=$x;
        $this->y=$y;
        }

    function setLabel($label) {
        $this->label=$label;
        }

    function getPoint() {
        return array("x"=>$this->x,
                     "y"=>$this->y,
                     "label"=>$this->label);
```

```
        }
}
// "$label" объявлена, но не установлена
$p1=new Point2D(1.233, 3.445);
print_r(get_object_vars($p1));
$p1->setLabel("point #1");
print_r(get_object_vars($p1));
?>

/* Будет выведено:
Array
 (
     [x] => 1.233
     [y] => 3.445
     [label] =>
 )
 Array
 (
     [x] => 1.233
     [y] => 3.445
     [label] => point #1
 ) */
```

`bool` **`method_exists`**`(object` **`object`**`, string` **`method_name`**`)`

Функция возвращает `true`, если метод **`method_name`** был объявлен для **`object`** . В противном случае функция возвращает `false`.

`bool` **`property_exists`**`(mixed` **`class`**`, string` **`property`**`)`

Функция проверяет, существует ли свойство **`property`** в определенном классе **`class`** (и доступно ли оно из текущей области видимости). Возвращает `true`, если свойство существует, `false` – если оно не существует или `null` в случае ошибки. В противоположность `isset()`, `property_exists()` возвращает `true` даже если свойство имеет значение `null`.

`array` **`get_declared_classes`**`()`

Эта функция возвращает массив имен классов, объявленных в текущей итерации.

Учтите, что в зависимости от набора библиотек, собранных с **PHP**, может варьироваться число дополнительных классов, так как дополнительные расширения могут объявлять свои классы.

Это также означает, что вы не сможете использовать имена их классов для своих.

ПРИМЕР 4.1.5

```
<?php
```

```
print_r(get_declared_classes());
?>

/* Пример выведет что-то подобное:
Array
(
    [0] => stdClass
    [1] => __PHP_Incomplete_Class
    [2] => Directory
)
*/
```

`string` **`get_parent_class`**`(mixed` **`obj`**`)`

Функция возвращает имя класса базового для класса, экземпляром которого является **obj**. Если **obj** является строкой, функция возвращает имя класса базового для класса с этим именем.

ПРИМЕР 4.1.6

```
<?php
class dad {
   function __construct() { return(true); }
}

class child extends dad {
   function __construct() {
      echo "Я - дочерний класс ".get_parent_class($this);
      }
}

class child2 extends dad {
   function __construct() {
      echo "И я дочерний класс ".get_parent_class('child2');
      }
}

$foo=new child();
$bar=new child2();
?>
/* вывод:
Я - дочерний класс dad
И я дочерний класс dad
*/
```

Как видите, при помощи данных функций, действительно можно много узнать об объектах и их классах, использующихся в ваших скриптах.

Понятно, что область применения данных функций, чаще всего,

это отладка или скрипты каких-то административных страниц сайтов.

ДОМАШНЕЕ ЗАДАНИЕ.

В прошлом уроке домашним заданием было создать класс «Органайзер».

Сегодня предлагаем Вам разработать систему классов, которая станет основой для построения редактора файлов, необходимых для разработки сайтов: **html**, **css**, **php**.

Каждый файл представляет собой объект с определенным набором свойств, присущих определенному типу. Например, **html** файл имеет DOCTYPE, кодировку, язык, связанную таблицу стилей, связанный файл скриптов **JavaScript**.

Кроме того, будучи достаточно разными, файлы, тем не менее, обладают определенной общностью, поэтому классы, реализующие свойства и методы определенного типа могут быть унаследованы от некоторого общего родителя.

Таким образом, Вам необходимо создать скрипт, который будет давать возможность пользователю создать некоторый файл нужного типа и определить его свойства, наполнить его некоторым контентом, иметь возможность связывать эти файлы между собой. В идеале, конечно, этот файл нужно бы сохранить в файловой системе сервера и предоставить пользователю возможность дальнейшего управления этими файлами. Но, мы пока еще не можем этого сделать, поэтому данное задание будет своего рода заготовкой, которую мы доработаем после **7**-го урока, т.е. когда изучим принципы работы с файловой системой сервера.

Результат должен представлять собой файл (или набор файлов), в котором будет реализован абстрактный родительский класс файла и наследованные от него классы специфичных файлов, а также управляющий скрипт, при помощи которого мы сможем создавать экземпляры объектов и задавать нужные значения их свойствам.

ИТОГИ.

Итак, мы завершили в данном уроке рассмотрение концепции **ООП** и её реализации в **PHP 5**.

Современные принципы web-разработки на **PHP** совершенно

немыслимы без использования классов и объектов. В этом мы с Вами убедимся в будущем, когда приступим к рассмотрению более сложных тем и, главное, профессионального применения программирования на **PHP** на примере законченных проектов, использующихся в современном web-строительстве. Таких как шаблонизатор **smarty** и разного рода **CMS**.

Напоследок хотелось бы отметить некоторую парадоксальность использования классов в **PHP** по сравнению с системным программированием.

Когда программа генерирует на основе классов объекты, необходимые ей для работы, то они помещаются в оперативную память и живут там ровно столько, сколько длится сеанс работы данного приложения, что позволяет в том числе экономить вычислительные ресурсы, ибо создание всей объектной структуры происходит один раз.

В Web-программировании все не так, поскольку, как мы знаем, скрипт «живёт» считанные секунды, после чего, выдав web-серверу результат своей работы, «умирает», освобождая всю занятую оперативную память. Т.е., вся структура объектов, необходимая скрипту всякий раз при обращении к нему генерируется заново.

Тем не менее, несмотря на это обстоятельство, использование ООП становится все более и более ёмким. В данной связи хочется дать только один совет: создавайте свои объектные структуры оптимально и рационально, чтобы не заставлять сервер делать лишнюю работу по генерации неиспользуемых объектов.

Желаем удачи в использовании ООП.