**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ «МЭИ»**

**ИНСТИТУТ ИНФОРМАЦИОННЫХ И ВЫЧИСЛИТЕЛЬНЫХ ТЕХНОЛОГИЙ**

**Кафедра ВМСС**

Тема: «Объектно-ориентированное программирование в PHP. Примеры»

Реферат

по дисциплине «WEB-технологии»

|  |  |  |
| --- | --- | --- |
|  |  | Выполнил:  Студент группы А-08-19 Балашов С.А.  Принял:  асс. Мишин А.А. |

Москва, 2021

Содержание

1. **Введение 3**
2. **Организация библиотек 4**
   1. Подключение файла библиотеки 4
   2. Разрешение конфликтов имен 5
   3. Автоматическая загрузка классов 6
   4. Главный файл скрипта 7
   5. Интерфейс библиотеки 8
   6. Наследование и расширение модулей 10
3. **Классы и сокрытие данных 11**
   1. Класс как тип данных 11
   2. Создание нового класса 11
   3. Работа с классами 12
   4. Инициализация и разрушение 14
   5. Права доступа к членам класса 16
4. **Наследование и виртуальные методы 19**
   1. Расширение класса 19
   2. Полиморфизм 21
   3. Интерфейсы 25
5. **Обработка ошибок и исключения 27**
   1. Ошибки 27
   2. Исключения 28
6. **Итераторы, массивы 30**
   1. Неявный доступ к классам и методам 30
   2. Итераторы 31
   3. Виртуальные массивы 31
7. **Заключение 34**
8. **Список использованной литературы 35**

# Введение

Большинство современных языков программирования используют методологию объектно-ориентированного программирования (ООП). Она позволяет представить программу в виде совокупности объектов, каждый из которых является частью определенного класса. В свою очередь классы образуют иерархию наследования.

Такой подход позволяет проще работать с крупными проектами за счет разбиения программы на своеобразные модули, каждый из которых выполняет определенную задачу. Несмотря на снижение производительности программ при использовании ООП из-за нескольких слоев абстракции, инкапсуляции и динамического связывания методов, данный подход является на текущий момент самым популярным благодаря схожести с естественными языками.

В этом реферате я рассмотрю ООП в языке PHP - скриптовом языке общего назначения, входящий в Топ-10[[1]](#footnote-1) популярных языков программирования на момент ноября 2021 года.

# Организация библиотек

## Подключение файла библиотеки

Один из самых важных навыков в программировании - грамотное разделение кода программ на относительно независимые группы функций - *библиотеки.* Библиотека может храниться в одном или нескольких файлах; случаются также ситуации, когда несколько библиотек хранятся в одном документе для ускорения загрузки.

Библиотека атомарна: достаточно в одном месте программы написать код ее подключения, и ниже этого места можно пользоваться ее содержимым.

Введем ряд правил, которых будем придерживаться на протяжении всего реферата. Все необходимые библиотеки будем хранить в одном каталоге - lib. Этот каталог находится в каталоге документов сервера (его имя всегда доступно через getenv(“DOCUMENT\_ROOT”)) и для него запрещен просмотр через браузер. Запрета можно добиться при использовании сервера Apache, создав в каталоге файл .htaccess следующего содержания (листинг 2.1).

### Листинг 2.1. Файл lib/.htaccess

deny from all

Теперь нужно, чтобы при вызове

require\_once "library\_name.php"

происходил поиск этого файла в указанном каталоге. Для этого следует изменить внутреннюю переменную PHP с именем include\_path, которая задается в файле php.ini.

Поскольку изменять php.ini в большинстве случаем нельзя, в скрипте, которому требуется библиотека, воспользуемся функцией ini\_set():

# добавить путь поиска библиотек ini\_set("include\_path", getenv("DOCUMENT\_ROOT")."/lib");

#...

#теперь можно подключать: require\_once “library\_name.php";

Лучше всего выделить код из листинга 2.2 в отдельный файл, который подключается в начале работы. Создадим файл lib/config.php.

### Листинг 2.2. Файл lib/config.php

<?php

if (!defined("PATH\_SEPARATOR"))

define("PATH\_SEPARATOR", getenv("COMSPEC")? ";" : ":");

ini\_set("include\_path", ini\_get("include\_path").PATH\_SEPARATOR.dirname(\_\_FILE\_\_));

?>

Теперь появилась возможность писать фрагменты такого вида:

require\_once getenv("DOCUMENT\_ROOT")."/lib/config.php";

...

require\_once “library\_name.php";

## Разрешение конфликтов имен

Пускай была написана функция length(), вычисляющую количество элементов массива, и используете ее в своей программе. Через некоторое время понадобилось подключить библиотеку стороннего разработчика, и выясняется, что в ней тоже есть функция length(), но уже для определения длины строки. Возникает конфликт имен.

На ранних стадиях создания PHP разработчики не использовали существующий сейчас стандарт и называли функции односложно и не всегда корректно отображая их суть: current(), key(), sort(), range() и т. д. По мере разработки языка число функций настолько выросло, что давать им подобные названия стало нецелесообразно. Возникла высокая вероятность отказа работы пользовательских программ из-за конфликта имен.

Одно из временных решений - добавлять в имена функций некоторый префикс, отвечающий их назначению. Так появились array\_keys(), array\_merge(), array\_splice() и т. д. В их именах применяется префикс array\_, свидетельствующий о том, что речь идет о работе с массивами. Такой префикс гарантирует в определенной степени уникальность имени, зато сильно удлиняет название функции.

Применение префикса обладает большим недостатком. Если понадобится его поменять, придется изменить в программе каждое имя функции и переменной.

Существует решение этой проблемы. Вместо того чтобы использовать префиксы в именах функций, можно поместить все объекты программы в так называемое *пространство имен*.

Пространство имен (namespace) - это имеющий имя фрагмент программы, содержащий в себе функции, переменные, константы и другие именованные сущности. Для получения "извне" доступа к идентификатору из некоторого пространства имен служит синтаксис:

имяПространстваИмен::имяИдентификатора

Если осуществляется работа внутри некоторого пространства имен, не обращаясь к другим, то нет необходимости явно его указывать: подразумевается, что вначале идентификаторы ищутся в текущем пространстве.

Удобство такого способа в том, что он позволяет решить проблему именования идентификатора. Представим, что есть два пространства имен: main (основная программа) и lib (сторонняя библиотека). В обеих определена функция length(). При этом доступ к одной функции будет выглядеть как lib::length(), а к другой - main::length(). Причем для кода, который находится в пространстве имен main, префикс main:: можно не указывать. Представим программу на "псевдоязыке", поддерживающем работу с пространствами имен (к таким языкам относятся C++, C#, Java и т.д.):

namespace lib { function length($str) { ... }

}

namespace main { function length($arr) { ... }

...

echo length($a); # вызывается main::length()

echo lib::length($s); # вызывается lib::length()

}

## Автоматическая загрузка классов

Из предыдущих пунктов можно заметить, что прежде, чем использовать какой-то модуль, его необходимо подключить. Если библиотек много, их подключение может оказаться долгим и муторным процессом. В PHP существует средство, позволяющее загружать классы автоматически: как только программа пытается обратиться к несуществующему классу, вызывается функция spl\_autoload\_register().

Напишем функцию для загрузки классов MyClass1 и MyClass2 из файлов *MyClass1.php* и *MyClass2.php* соответственно:

<?php

spl\_autoload\_register(function ($class\_name) {

include $class\_name . '.php';

});

$obj  = new MyClass1();

$obj2 = new MyClass2();

?>

Функция spl\_autoload\_register() позволяет зарегистрировать необходимое количество автозагрузчиков для автоматической загрузки классов и интерфейсов, если они в настоящее время не определены.

## Главный файл скрипта

Обычно, при написании больших скриптов стараются разбить их на некоторое число независимых библиотек (модулей). Хорошим тоном считается перенос основного кода программы в библиотеки так, чтобы головной файл скрипта лишь подключал их и вызывал одну из функций оттуда. Это позволяет более удобно отлаживать программы.

Например, если URI запроса выглядит как /path/to/view.php, файл view.php может содержать:

<?php

require\_once "lib/config.php";

require\_once "System/MainLib.php";

System\_MainLib::doView();

?>

Как видите, основная работа происходит в функции System\_MainLib::doView(), а головной файл лишь занимается ее подключением.

Чтобы выйти из глобальной области видимости, в PHP существуют функции. Таким образом, рекомендуется основной код программы размещать внутри функции, а к глобальным переменным иногда обращаться явным способом: либо через инструкцию global, либо через массив $GLOBALS.

Однако создать функцию и тут же ее использовать - решение не очень элегантное. Во-первых, так не решается проблема констант, которые могут понадобиться главному файлу (определять константы при помощи define() в глобальной области видимости, опять же, нежелательно). Во-вторых, опять "засоряется" глобальная область видимости, но на этот раз не переменными, а функциями.

Решение проблемы известно: следует заключить основной код программы в область видимости (листинг 2.4).

### Листинг 2.4. Файл script.php

<?php

class script {

# Основная функция программы. Запускается при старте скрипта.

static function main() {

$start = self::microtime();

sleep(1);

echo "Параметр test: ".@$\_REQUEST['test'];

$end = self::microtime();

echo "<hr>Программа работала ".sprintf("%.2f", $end-$start)." с";

}

#double microtime()

#Возвращает текущее время в секундах в виде ДЕЙСТВИТЕЛЬНОГО

#числа (с долями секунды). Заменяет стандартную microtime(),

#возвращающую массив, с которым не очень удобно работать.

private static function microtime() {

$t = explode(" ", microtime());

return $t[0]+$t[1];

}

}

if (!defined("DONT\_CALL\_MAIN")) script::main(); ?>

## Интерфейс библиотеки

Интерфейс модуля - это то, как он "виден" извне. Иными словами, это набор функций, переменных и констант, доступных в подключающей библиотеку программе.

Ранее функции помещались в области видимости, чтобы они сразу же были доступны внешним программам. Это не всегда желательно: ведь существуют вспомогательные и служебные функции для внутреннего использования.

Чтобы исключить функцию или переменную из интерфейса библиотеки, нужно написать перед ее объявлением ключевое слово private (листинг 2.5). Слово public, наоборот, подчеркивает, что идентификатор общедоступен, "открыт" (режим по умолчанию).

### Листинг 2.5. Файл t\_private.php

<?php

class test {

#Скрытая переменная.

private static $v = 10;

#Общедоступная функция.

public static function pub() {

echo "public (v=".self::$v.")<br>";

self::pri("вызов изнутри класса");

}

#Скрытая функция.

private static function pri($from) {

echo "private $from<br>";

}

}

test::pub();

test::pri("снаружи");

?>

Результат работы данного скрипта выглядит так:

Public (v=10)

private вызов изнутри класса

Fatal error: Call to private method test::pri() from context '' in c:\\savva\php\src\libraries\t\_private.php on line 13

Как видите, вызов функции pri() из функции pub() прошел штатно, а вызов из основной программы привел к ошибке.

Рекомендуется объявлять при помощи private как можно больше идентификаторов, оставляя доступным лишь самое необходимое.

## Наследование и расширение модулей

Предположим, есть некоторый модуль (возможно, написанный другим программистом) с некоторым интерфейсом. Существует задача добавить в этот модуль еще несколько функций, чтобы они были доступны наравне с имеющимися.

В этом случае можно воспользоваться наследованием (листинг 2.6).

### Листинг 2.6. Файл lib/MyFileFindExt.php

<?php

require\_once "MyFileFind.php";

class MyFileFindExt extends MyFileFind {

# Переопределяется имеющаяся функция.

public function readdir($dir) {

echo "readdir($dir) called\n";

# Вызывается исходная функция из MyFileFind.

return parent::readdir($dir);

}

# Новая функция.

public function readcurdir() {

return self::readdir(".");

}

}

# Печатается диагностическое сообщение (для примера).

echo "File ".\_\_FILE\_\_." loaded.\n";

?>

При указании строки "class MyFileFindExt extends MyFileFind", создается класс, содержащий все те же функции, переменные и константы, что и MyFileFind, и еще несколько. Класс, от которого происходит наследование (MyFileFind), называется базовым, а новый класс (MyFileFindExt) - производным.

# Классы и сокрытие данных

## Класс как тип данных

Ключевым понятием ООП является *класс*. Класс можно рассматривать как тип некоторой переменной. Переменная класса (далее: объект класса) обычно имеет набор свойств (значений различных типов) и операций, которые могут быть с ним проведены. Свойства и методы класса часто называют его членами.

Например, можно рассмотреть тип int как класс. Тогда переменная этого "класса" будет обладать одним свойством (ее целым значением), а также набором методов (сложение, вычитание, инкремент и т. д.). При этом методы выглядят как арифметические операторы +, -, ++ и т. д.

В языке C++ можно было бы объявить новый тип Int именно таким образом путем перегрузки. Однако в PHP дело обстоит немного сложнее: у программиста нет прав для переопределения стандартных операций (сложение, вычитание и т. д.) для объектов. Например, если бы возникла необходимость добавить в язык комплексные числа, в C++ это можно было сделать без особых затруднений, однако в PHP такое добавление не удастся.

Альтернативное решение состоит в том, чтобы везде вместо + и других операций использовать вызовы соответствующих функций, например, add(), которые бы являлись методами класса.

Подход к созданию классов, применяемый в объектно-ориентированных языках, называют инкапсуляцией. Данные, принадлежащие классу, сохраняются в его свойствах, доступ к которым тщательно ограничивается и предоставляется в основном при помощи специальных методов.

## Создание нового класса

Новый класс в программе описывается при помощи ключевого слова class. Внутри класса могут располагаться его свойства (переменные класса) и методы (функции-члены класса).

Опишем класс с именем Math\_Complex, объекты которого будут хранить комплексные числа (листинг 3.1). Этот класс пока поддерживает только сложение и вычитание.

### Листинг 3.1. Файл lib/Math/Complex.php

<?php

class Math\_Complex {

//Свойства: действительная и мнимая части.

public $re, $im;

//Метод: добавить число к текущему значению. Число задается

//своей действительной и мнимой частью.

function add($re, $im) {

$this->re += $re;

$this->im += $im;

} } ?>

Файл, приведенный в листинге 3.1, при своем включении не выполняет никаких действий. Его задача - добавить в программу новый класс с именем Math\_Complex. Можно заметить, что в этом отношении описание класса очень похоже на описание библиотеки: один файл - один класс.

Однако есть одно большое отличие. Если бы Math\_Complex был описан как библиотека (а переменные $re и $im - как static-члены), в программе имелся бы единственный экземпляр пары переменных $re и $im, с которой работала бы функция add(). В то же время, описание класса позволяет в скрипте создавать несколько объектов-экземпляров данного класса, у каждого из которых будет своя собственная пара переменных ($re, $im). В этом отношении класс напоминает библиотеку, способную к "размножению".

## Работа с классами

Предположим, что в программе каким-то образом уже описан некоторый класс. Так как класс - это, по сути, тип данных, должен существовать некоторый механизм для создания переменных, хранящих значение этого типа.

При создании переменных, имеющих пользовательский тип данных, применяется ключевое слово new, за которым следует имя класса:

$obj = new Math\_Complex;

Теперь $obj хранит все данные класса - в частности, содержит внутри себя отдельные значения $re и $im.

Каждый объект имеет свой собственный набор ассоциированных с ним свойств (значений, или переменных) и множество методов. Каждое свойство объекта доступно в программе по его имени. Можно присваивать значение свойству или получать его величину:

//Создается новый объект класса Math\_Complex.

$obj = new Math\_Complex;

//Присваивает значение свойствам $re и $im объекта $obj.

$obj->re = 6;

$obj->im = 101;

//Выводит значение свойства re объекта $obj.

echo $obj->re;

В PHP для вызова метода некоторого объекта используется оператор "стрелка" (листинг 3.2).

### Листинг 3.2. Файл call.php

<?php

//Подключение каталога библиотек в include\_path.

require\_once "lib/config.php";

//Загрузка класса.

require\_once "Math/Complex.php";

//Создается новый объект класса Math\_Complex.

$obj = new Math\_Complex;

//Присваивается начальное значение свойствам.

$obj->re = 16.7;

$obj->im = 101;

//Вызов метода add()с параметрами (18.09, 303) объекта $obj.

$obj->add(18.09, 303);

//Выводится результат:

echo "({$obj->re}, {$obj->im})"; ?>

Давайте посмотрим, что происходит, когда вызывается метод класса. Первым делом создается локальная переменная $this, которой присваивается то же значение, что было у $obj. То есть, в $this теперь хранится ссылка на объект, для которого вызывается метод. Далее PHP смотрит, какому классу принадлежит $obj (в данном случае это Math\_Complex) и находит функцию-член: Math\_Complex::add(). Функция вызывается, при этом $this, напомним, равен $obj. В итоге add() изменяет значения $obj->re и $obj->im.

Как видно, вызов метода некоторого объекта автоматически предоставляет ему доступ к свойствам этого объекта посредством специальной переменной $this. При этом $this не нужно нигде объявлять явно, она появляется сама собой. Данная техника - ключевая особенность ООП.

## Инициализация и разрушение

Для корректного создания объекта недостаточно просто использовать оператор new: потом приходится еще инициализировать свойства объекта ($re и $im). В листинге 3.3 представлена реализация класса комплексных чисел с инициализацией.

### Листинг 3.3. Файл lib/Math/Complex2.php

<?php

class Math\_Complex2 {

public $re, $im;

//Инициализация нового объекта.

function \_\_construct($re, $im) {

$this->re = $re; $this->im = $im;

}

//Добавляет к текущему комплексному числу другое. function add(Math\_Complex2 $y) {

$this->re += $y->re;

$this->im += $y->im;

}

// Преобразует число в строку (например, для вывода).

function \_\_toString() {

return "({$this->re}, {$this->im})";

} } ?>

Метод \_\_construct() - это *конструктор* класса. Он вызывается всегда, когда используется оператор new для объекта.

В данном примере конструктор принимает два параметра: действительную и вещественную часть комплексного числа. Листинг 3.4 показывает применение данного класса.

Листинг 3.4. Файл construct.php

<?php

require\_once "lib/config.php";

require\_once "Math/Complex2.php";

$a = new Math\_Complex2(314, 101);

$a->add(new Math\_Complex2(303, 6));

echo $a;

?>

Как и для обычных функций и методов, для конструкторов можно задавать параметры по умолчанию. Например, объявив его следующим образом:

function \_\_construct($re=0, $im=0) {

$this->re = $re;

$this->im = $im;

}

По аналогии с конструкторами обычно рассматриваются *деструкторы*. Деструктор - специальный метод объекта, который вызывается при уничтожении этого объекта (например, после завершения программы). Деструкторы обычно выполняют служебную работу - закрывают файлы, записывают протоколы работы, разрывают соединения.

В листинге 3.5 приведен класс с именем File\_Logger, в котором объявляется деструктор. Нет необходимости заботиться о "ручном" вызове close() в программе - PHP выполняет "завершающие" действия самостоятельно.

### Листинг 3.5. Файл lib/File/Logger.php

<?php

class File\_Logger {

public $f; // открытый файл

public $name // имя журнала

public $lines = array();

// накапливаемые строки

public $t;

//Создает новый файл журнала или открывает дозапись в конец

//существующего. Параметр $name - логическое имя журнала.

public function \_\_construct($name, $fname) {

$this->name = $name;

$this->f = fopen($fname, "a+");

$this->log("### \_\_construct() called!");

}

//Гарантированно вызывается при уничтожении объекта.

//Закрывает файл журнала.

public function \_\_destruct() { $this->log("### \_\_destruct() called!");

//Вначале выводятся все накопленные данные.

fputs($this->f, join("", $this->lines));

//Затем закрывается файл.

fclose($this->f);

}

//Добавляет в журнал одну строку. Она не попадает в файл сразу же,

//а записывается в буфер и остается там до вызова \_\_destruct(). public function log($str) {

//Каждая строка предваряется текущей датой и именем журнала.

$prefix = "[".date("Y-m-d\_h:i:s ")."{$this->name}] ";

$str = preg\_replace('/^/m',

$prefix, rtrim($str)); // Сохраняем строку.

$this->lines[] = $str."\n";

} } ?>

Рассмотрим, как может выглядеть использование данного класса (листинг 3.6).

### Листинг 3.6. Файл destr.php

<?php ##

require\_once "lib/config.php";

require\_once "File/Logger.php";

for ($n=0; $n<10; $n++) {

$logger = new File\_Logger("test$n", "test.log");

$logger->log("Hello!");

}

exit();

?>

## Права доступа к членам класса

До сих пор свойства и методы класса объявлялись без учета того, должны ли они быть доступны в программе, или же используются только для внутренних целей.

В PHP существуют три модификатора ограничения доступа: public, protected и private. Их можно указывать перед описанием метода или свойства класса.

* **Public**: открытый доступ

Члены класса, отмеченные ключевым словом public ("публичный", "открытый"), доступны для использования вне класса (например, из вызывающей программы). Пример:

class Hotel { public $exit;

public function escape() {

echo "Let's go through the {$this->exit}!";

}

}

$theLafayette = new Hotel();

$theLafayette->exit = "main wet wall"; // допустимо

$theLafayette->escape(); // допустимо

* **Private**: доступ только из методов класса

С использованием ключевого слова private ("личный", "закрытый") можно сделать члены касса "невидимыми" для вызывающей программы, будто бы их и нет. В тоже время, методы "своего" класса могут обращаться к ним без ограничений. Пример:

class Hotel { private $exit;

public function escape() {

$this->findWayOut(); // допустимо

echo "Let's go through the {$this->exit}!"; // допустимо

}

public function lock() { $this->exit = null; } private function findWayOut() {

$this->exit = "main wet wall"; // допустимо

}

}

$theLafayette = new Hotel(); $theLafayette->findWayOut(); $theLafayette->escape(); $theLafayette->exit = "hotel doors";

//Ошибка! Доступ закрыт!

//допустимо

//Ошибка! Доступ закрыт!

* **Protected**: доступ из методов производного класса

Модификатор protected ("защищенный") с точки зрения вызывающей программы выглядит точно так же, как и private: он запрещает доступ к членам объекта извне. Однако по сравнению с private он позволяет обращаться к членам не только из "своих" методов, но также и из методов производных классов (если используется наследование).

# Наследование и виртуальные методы

## Расширение класса

При помощи механизма *наследования* можно создавать новые типы данных не "с нуля", а взяв за основу некоторый, уже существующий, класс, который в этом случае называют базовым (base class). Получившийся же класс носит имя производного (derived class).

Наследование в ООП используется для нескольких различных целей:

* добавление в существующий класс новых методов и свойств или замена уже существующих;
* наследование в целях классификации и обеспечения однотипности поведения различных классов.

Дело в том, что новый, производный класс обладает теми же самыми "особенностями", что и базовый, и может использоваться везде вместо последнего. Например, рассмотрим базовый класс *Автомобиль* и производный от него - *БМВ*. Очевидно, что везде, где требуется объект типа *Автомобиль*, можно подставить и объект типа *БМВ* (но не наоборот). Создав еще несколько производных от *Автомобиля* классов (*Мерседес*, *Бентли*, *Газ* и т. д.), в ряде случаев можно работать с ними всеми однотипным образом, как с объектами типа *Автомобиль*, не вдаваясь в детали.

Пусть есть некоторый класс File\_Logger (см. п. 3) с определенными свойствами и методами. В листинге 4.1 приведен его код.

### Листинг 4.1. Файл lib/File/Logger.php

public $lines = array();

// накапливаемые строки

public $t;

public function \_\_construct($name, $fname) { $this->name = $name;

$this->f = fopen($fname, "a+");

}

public function \_\_destruct() { fputs($this->f, join("", $this->lines)); fclose($this->f);

}

public function log($str) {

$prefix = "[".date("Y-m-d\_h:i:s ")."{$this->name}] "; $str = preg\_replace('/^/m', $prefix, rtrim($str)); $this->lines[] = $str."\n";

} } ?>

Допустим, что действия этого класса не удовлетворяют требованиям, например, он выполняет большинство необходимых функций, но не реализует некоторые другие. Создадим новый класс File\_Logger\_Debug, как бы "расширяющий" возможности класса File\_Logger. Он будет добавлять ему несколько новых свойств и методов (листинг 4.2).

### Листинг 4.2. Файл lib/File/Logger/Debug.php

<?php

//Вначале подключается "базовый" класс.

require\_once "File/Logger.php";

//Класс, добавляющий в File\_Logger новую функциональность.

class File\_Logger\_Debug extends File\_Logger {

public function \_\_construct($fname) {

parent::\_\_construct(basename($fname), $fname);

}

public function debug($s, $level=0) { $stack = debug\_backtrace();

$file = basename($stack[$level]['file']); $line = $stack[$level][‘line'];

$this->log("[at $file line $line] $s");

} } ?>

Ключевое слово extends говорит о том, что создаваемый класс File\_Logger\_Debug является лишь "расширением" класса File\_Logger, и не более того. То есть File\_Logger\_Debug содержит те же самые свойства и методы, что и File\_Logger, но помимо них и еще некоторые дополнительные, "свои".

Теперь “часть" File\_Logger находится прямо внутри класса File\_Logger\_Debug и может быть легко доступна, наравне с методами и свойствами самого класса File\_Logger\_Debug.

Итак, видно, что, действительно, класс File\_Logger\_Debug является воплощением идеи "расширение функциональности класса File\_Logger".

## Полиморфизм

*Полиморфизм* (многоформенность) - одно из интересных следствий идеи наследования. Полиморфность - это способность объекта использовать методы не своего собственного класса, а производного, даже если на момент определения базового класса производный еще не существует.

С использованием полиморфизма можно писать классы-шаблоны, реализующие некоторую функциональность лишь частично, и лишь в той степени, в которой она им самим "известна". В дальнейшем, создавая производные классы, можно уточнить остальную часть кода, специфичную для приложения.

Пусть где в программе имеется класс Shape, соответствующий некоторой геометрической фигуре. Программа должна уметь выполнять два действия:

* перемещать фигуры при вызове метода moveBy();
* увеличивать или уменьшать размер фигуры с вызовом метода resizeBy().

Каждому действию соответствует один метод, в данном случае - это методы moveBy() и resizeBy(). Также каждая фигура обладает такими свойствами:

* координатами (свойства $x, $y);
* текущим масштабом ($scale).

В листинге 4.3 показано определение базового класса Shape, удовлетворяющее описанным выше условиям.

### Листинг 4.3. Файл shapes/Shape.php

<?php

class Shape {

private $x=0, $y=0, $scale=1.0;

//Конструктор класса. Отображает фигуру на экране.

public function \_\_construct() {

$this->show();

}

//Деструктор класса. Стирает фигуру с экрана.

public function \_\_destruct() {

$this->hide();

}

public final function moveBy($dx, $dy) {

$this->hide();

$this->x += $dx; $this->y += $dy;

$this->show();

}

//Изменить масштаб отображения фигуры.

public final function resizeBy($coef) {

$this->hide(); $this->scale \*= $coef; $this->show();

}

//Методы возвращают координаты центра и масштаб.

public final function getCoord() { return array($this->x, $this->y); } public final function getScale() { return $this->scale; }

protected function hide() {

die(“действие неизвестно");

}

protected function show() {

die("действие неизвестно");

} } ?>

Для гарантии того, что все геометрические фигуры будут вести себя одинаково, объявляются публичные методы финальными (final). Таким образом, их уже нельзя будет переопределить в производных классах, а значит, фигуры не смогут "отойти от принятых канонов".

При описании класса Shape не делается никаких предположений о типе фигуры. Чтобы стереть и нарисовать еще неизвестную фигуру используются *виртуальные* *методы*.

Виртуальным называют метод, который может переопределяться в производном классе. А в этом случае функции show() и hide() являются виртуальными, и даже более того: в классе Shape неизвестно, как они должны быть "устроены", потому что все еще нет информации о типе фигуры. Таким образом, вызывать виртуальные методы Shape бессмысленно, что подчеркивается запуском встроенной функции die() в них (см. листинг 4.3).

Раз в базовом классе Shape виртуальные методы show() и hide() "вырождены" и являются абстрактными, обязательно нужно переопределить их в производном классе (листинг 4.4).

### Листинг 4.4. Файл shapes/Circle.php

<?php

require\_once "ShapeA.php";

class Circle extends Shape {

//Радиус круга в масштабе 1:1.

private $radius;

//Создается новый объект-круг с указанием радиуса.

public function \_\_construct($radius=100) {

$this->radius = $radius; parent::\_\_construct();

}

//Отображает круг на экране.

public function show() {

list ($x, $y) = $this->getCoord();

$radius = $this->radius \* $this->getScale();

// \*Код прорисовки круга\* ($x, $y, $radius). echo "Рисуется круг: ($x, $y, $radius)<br>";

}

// Стирает фигуру с экрана. public function hide() {

list ($x, $y) = $this->getCoord();

$radius = $this->radius \* $this->getScale();

// \*Код стирания круга\* ($x, $y, $radius). echo "Стирается круг: ($x, $y, $radius)<br>";

} } ?>

Рассмотрим класс из листинга 4.4 подробнее.

Любой объект-круг является также и объектом-фигурой, а потому должен наследовать методы и свойства класса Shape. Поэтому код базового класса Shape подключается в самом начале файла, а также объявляется Circle производный класс от Shape.

У круга, помимо свойств, присущих фигуре, есть и свои собственные данные: это его радиус. Создается свойство $radius, в котором он будет храниться. Чтобы подчеркнуть, что радиус - свойство сугубо служебное и не может быть доступно извне, оно объявляется как закрытое (private).

Круг имеет свой собственный конструктор, который вызывается в момент создания объекта. При создании указывается радиус круга. Фактически у каждого типа фигуры будут свои собственные конструкторы с различающимися списками параметров. Конструктор обычно не наследуется.

Обязательно должен быть вызван конструктор базового класса Shape, в противном случае фигура не будет проинициализирована! Это делается явным образом: parent::\_\_construct().

Главное преимущество, которое дает наследование и полиморфизм, - это несравненная легкость создания новых классов, ведущих себя сходным образом с уже существующими. Добавить в программу новую геометрическую фигуру (например, квадрат) крайне просто: достаточно лишь написать ее класс, сделав его производным от Shape. После этого любая программа, которая могла работать с кругами, начнет работать и с квадратами. Единственное изменение, которое придется внести в код, - это создание объекта-квадрата.

Дадим определения терминам "абстрактный класс" и "абстрактный метод”.

Абстрактный метод нельзя вызвать, если он не был переопределен в производном классе. Собственно, написанием функции Shape::show() и помещением в нее вызова die(), гарантируется, что она обязательно будет переопределена в производном классе.

Объект абстрактного класса невозможно создать.

Любой класс, содержащий хотя бы один абстрактный метод, сам является абстрактным.

Специально для того, чтобы автоматически учесть эти особенности, в объектноориентированных языках программирования Java и PHP введено ключевое слово - модификатор abstract. Можно объявить класс или метод как abstract, и тогда контроль за их некорректным использованием возьмет на себя сам PHP.

Абстрактные классы можно использовать только для одной цели: создавать от них производные. В листинге 4.5 приведен все тот же самый класс Shape, но только теперь используется ключевое слово abstract там, где это необходимо по логике.

### Листинг 4.5. Файл shapes/ShapeA.php

<?php

abstract class Shape {

private $x=0, $y=0, $scale=1.0; public function \_\_construct() {

$this->show();

}

public function \_\_destruct() { $this->hide();

}

public final function moveBy($dx, $dy) { $this->hide();

$this->x += $dx; $this->y += $dy; $this->show();

}

public final function resizeBy($coef) { $this->hide();

$this->scale \*= $coef; $this->show();

}

public final function getCoord() { return array($this->x, $this->y); } public final function getScale() { return $this->scale; }

// Абстрактные методы.

abstract protected function hide(); abstract protected function show();

} ?>

Если случайно будет пропущено ключевое слово abstract в заголовке класса Shape, PHP напомнит об этом сообщением о фатальной ошибке:

Fatal error: Class Shape contains 2 abstract methods and must therefore be declared abstract (Shape::hide, Shape::show)

## Интерфейсы

Интерфейс (interface) представляет собой обычный абстрактный класс, но только в нем не может быть свойств, и не определены тела у методов. Фактически, некоторый интерфейс указывает лишь список методов, их аргументы и модификаторы доступа (обычно только protected и public). Допускается также описание констант внутри интерфейса (ключевое слово const).

Класс, наследующий некоторый интерфейс, обязан содержать в себе определения всех методов, заявленных в интерфейсе. Если хотя бы один из методов не будет реализован, невозможно будет создать объект класса: возникнет ошибка.

Главное достоинство заключается в том, что класс может реализовывать (наследовать) сразу несколько интерфейсов. Для "привязки" интерфейсов к классу используется ключевое слово implements:

interface IWorldObject {

public function getCoord(); // тело не указывается!

}

interface IVehicle {

public function getNumWheels(); // возвращает число колес

}

class Zaporojets implements IVehicle, IWorldObject {

public function getCoord() { ... }

public function getNumWheels() { ... }

}

Интерфейсы часто используются как средство классификации объектов в программе. Для каждого абстрактного типа объекта из предметной области создается собственный интерфейс, а затем, при описании новых классов, указывается, какие интерфейсы они наследуют - иными словами, как их можно классифицировать.

# Обработка ошибок и исключения

## Ошибки

Термин "ошибка" имеет три различных значения:

1. Ошибочная ситуация - непосредственно факт наличия ошибки в программе.

2. Внутреннее сообщение об ошибке ("внутренняя ошибка"), которую выдает PHP в ответ на различные неверные действия программы.

3. Пользовательское сообщение об ошибке ("пользовательская ошибка"), к которой причисляются все сообщения или состояния, генерируемые и обрабатываемые самой программой.

Далее под ошибкой будет подразумеваться некоторая информация о ней. В простейшем случае эта информация включает в себя текст диагностического сообщения, но могут также уточняться и дополнительные данные, например, номер строки и имя файла, где возникла ошибочная ситуация. Если в программе возникла ошибочная ситуация, необходимо принять решение, что же в этом случае делать. Код, который этим занимается называют кодом восстановления после ошибки, а запуск этого кода - восстановлением после ошибки. Рассмотрим пример:

$f = @fopen("spoon.txt", "r");

if (!$f) return;

Здесь код восстановления - это инструкция if, которая явно обрабатывает ситуацию невозможности открытия файла. Обратите внимание, что оператор @ используется перед fopen(), чтобы не получить диагностическое сообщение от самого PHP.

Ошибки по своей "серьезности" можно подразделить на два больших класса:

* серьезные ошибки с невозможностью автоматического восстановления. Например, если осуществляется попытка открыть несуществующий файл, то далее обязательно должны указать, что делать, если это не удастся: ведь записывать или считывать данные из неоткрытого файла нельзя;
* "несерьезные" (нефатальные) ошибки, восстановление после которых не требуется, например, предупреждения (warnings), уведомления (notices), а также отладочные сообщения (debug notices). Обычно в случае возникновения такого рода ошибочных ситуаций нет необходимости предпринимать что-то особенное и нестандартное, вполне достаточно просто сохранить где-нибудь информацию об ошибке (например, в файле журнала).

## Исключения

Механизм обработки исключений - это технология, позволяющая писать код восстановления после серьезной ошибки в удобном для программиста виде. С применением исключений перехват и обработка ошибок, наиболее слабая часть в большинстве программных систем, значительно упрощается.

Исключение - это некоторое сообщение об ошибке вида "серьезная". При своей генерации оно автоматически передается в участок программы, который лучше всего "осведомлен", что же следует предпринять в данной конкретной ситуации. Этот участок называется обработчиком исключения.

Любое исключение в программе представляет собой объект некоторого класса, создаваемый, как обычно, оператором new. Этот объект может содержать различную информацию, например, текст диагностического сообщения, а также номер строки и имя файла, в которых произошла генерация исключения. Допустимо добавлять и любые другие параметры. Рассмотрим пример вызова обработчика (листинг 5.1).

### Листинг 5.1. Файл simple.php

<?php

echo "Начало программы.<br>";

try {

// Код, в котором перехватываются исключения. echo "Все, что имеет начало…<br>";

//Генерируется исключение. throw new Exception("Hello!");

echo "...имеет и конец.<br>";

}catch (Exception $e) {

//Код обработчика.

echo " Исключение: {$e->getMessage()}<br>";

}

echo "Конец программы.<br>";

?>

В листинге 5.1 приведен пример базового синтаксиса конструкции try...catch, применяемой для работы с исключениями.

Код обработчика исключения помещается в блок инструкции catch (в переводе с английского - "ловить").

Блок try (в переводе с английского - "попытаться") используется для того, чтобы указать в программе область перехвата. Любые исключения, сгенерированные внутри нее (и только они), будут переданы соответствующему обработчику.

Инструкция throw используется для генерации исключения. Любое исключение представляет собой обычный объект PHP, который создается в операторе new.

Стоит обратить внимание на аргумент блока catch. В нем указано, в какую переменную должен быть записан "пойманный" объект-исключение перед запуском кода обработчика. Также обязательно задается тип исключения - имя класса. Обработчик будет вызван только для тех объектов-исключений, которые совместимы с указанным типом (например, для объектов данного типа).

# Итераторы, массивы

## Неявный доступ к классам и методам

В сложных приложениях приходится встречаться с ситуациями, когда одно из имен (или список аргументов) хранится в некоторой переменной, и в программе нельзя явно указать ее значение. Такое имя метода, класса или даже список аргументов функции называют неявным.

Листинг 6.1 иллюстрирует неявный вызов метода add() для объекта $a класса Math\_Complex2.

### Листинг 6.1. Файл impl\_meth.php

<?php

require\_once "lib/config.php";

require\_once "Math/Complex2.php";

$addMethod = "add";

$a = new Math\_Complex2(101, 303);

$b = new Math\_Complex2(0, 6);

// Вызывается метод add() неявным способом.

call\_user\_func(array(&$a, $addMethod), $b);

echo $a;

?>

Для того чтобы передать некоторому методу аргументы, хранящиеся в том или ином списке, в PHP существует всего одно средство - функция call\_user\_func\_array(). Она принимает два параметра: первый - это имя функции, а второй - массив, хранящий ее аргументы.

Вот как может выглядеть вызов функции test(), аргументы которой хранятся в массиве:

$args = array(101, 6);

$result = call\_user\_func\_array("test", $args);

А так вызывается метод test() некоторого объекта $obj:

$result = call\_user\_func\_array(array(&$obj, "test"), $args);

Для вызова статического метода вместо объекта необходимо указать строковое имя класса:

$result = call\_user\_func\_array(array("ClassName", "test"), $args);

Инстанциирование (instantiate) - это термин ООП, который означает "создание объекта некоторого класса". Инстанциировать класс - то же самое, что создать экземпляр (объект) этого класса.

Перейдем к вопросу о том, как создать объект некоторого класса, если имя этого класса задано неявно, например, содержится в переменной. Листинг 6.2 показывает, как поступать в таком случае.

### Листинг 6.2. Файл inst.php

<?php

require\_once "lib/config.php";

require\_once "Math/Complex2.php";

//Пусть имя класса хранится в переменной $className. $className = "Math\_Complex2";

//Создается новый объект.

$obj = new $className(6, 1); echo "Созданный объект: $obj"; ?>

## Итераторы

Итератор - это объект, класс которого реализует встроенный в PHP интерфейс Iterator. Он позволяет программе решать, какие значения необходимо подставлять в переменные инструкции foreach при ее работе и в каком порядке это делать.

Любой объект, который хочет переопределить стандартное поведение инструкции foreach, должен реализовывать встроенный в PHP интерфейс IteratorAggregate. Интерфейс определяет единственный метод - getIterator(), который должен создать объект-итератор.

В дальнейшем все решения о том, какие значения участвуют в переборе и в каком порядке их необходимо возвращать, принимает уже итератор.

## Виртуальные массивы

PHP позволяет создавать объекты, доступ к которым производится в соответствии с синтаксисом управления массивами PHP. Иными словами, можно использовать оператор [] для переменной-объекта, как будто работаете с обычным ассоциативным массивом. При этом возможно применение и обычного оператора -> для доступа к свойствам и методам объекта.

Если нужно указать интерпретатору, что к объекту некоторого класса возможно обращение, как к массиву, то придется использовать встроенный в PHP интерфейс ArrayAccess при описании соответствующего класса. Кроме того, необходимо определить тела методов, описанных в этом интерфейсе

Рассмотрим пример использования интерфейса ArrayAccess (листинг 6.3).

### Листинг 6.3. Файл array.php

<?php

class InsensitiveArray implements ArrayAccess {

//Здесь будет храниться массив элементов в нижнем регистре. private $a = array();

//Возвращает true, если элемент $offset существует. public function offsetExists($offset) {

$offset = strtolower($offset); // переводим в нижний регистр $this->log("offsetExists('$offset')");

return isset($this->a[$offset]);

}

//Возвращает элемент по его ключу.

public function offsetGet($offset) { $offset = strtolower($offset); $this->log("offsetGet('$offset')"); return $this->a[$offset];

}

//Устанавливает новое значение элемента по его ключу. public function offsetSet($offset, $data) {

$offset = strtolower($offset); $this->log("offsetSet('$offset', '$data')"); $this->a[$offset] = $data;

}

//Удаляет элемент с указанным ключом.

public function offsetUnset($offset) { $offset = strtolower($offset); $this->log("offsetUnset('$offset')"); unset($this->array[$offset]);

}

// Служебная функция для демонстрации возможностей. public function log($str) {

echo "$str<br>";

}

}

// Проверка.

$a = new InsensitiveArray();

$a->log("## Устанавливаются значения (оператор =)."); $a['php'] = 'There is more than one way to do it.';

$a['pHp'] = 'Это значение должно переписаться поверх предыдущего.'; $a->log("## Получаем значение элемента (оператор [])."); $a->log("<b>значение:</b> '{$a['PHP']}'");

$a->log("## Проверяется существование элемента (оператор isset())."); $a->log("<b>exists:</b> ".(isset($a['Php'])? "true" : "false")); $a->log("## Уничтожается элемент (оператор unset())."); unset($a['phP']);

?>

Результат работы данного кода выглядит примерно так:

##Устанавливаются значения (оператор =). offsetSet('php', 'There is more than one way to do it.')

offsetSet('php', 'Это значение должно переписаться поверх предыдущего.')

##Получается значение элемента (оператор []).

offsetGet('php')

значение: 'Это значение должно переписаться поверх предыдущего.' ## Проверяется существование элемента (оператор isset()). offsetExists('php')

exists: true

## Уничтожается элемент (оператор unset()). offsetUnset('php')

# Заключение

В этой работе было рассмотрено применение методологии ООП в скриптовом языке PHP. Было показано как работать с библиотеками, использовать классы и наследовать их, понятие полиморфизма, обработку ошибок и исключений, итераторы и виртуальные методы. Все сведения были подкреплены примерами кода, чтобы нагляднее демонстрировать предмет обсуждения. На текущий момент ООП - самая популярная парадигма в программировании и разработчику важно знать ее особенности и уметь её применять.

# Список использованной литературы

1. Wikipedia. PHP [Электронный ресурс]: свободная энциклопедия - /Wikipedia. - Электронные данные. Режим доступа: URL.: [https://ru.wikipedia.org/wiki/PHP#Объектно-ориентированное\_программирование](https://ru.wikipedia.org/wiki/PHP#%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), свободный - (дата обращения 25.11.2021)
2. Wikipedia. Объектно-ориентированное программирование [Электронный ресурс]: свободная энциклопедия - /Wikipedia. - Электронные данные. Режим доступа: URL.: [https://ru.wikipedia.org/wiki/Объектно-ориентированное\_программирование#Производительность\_объектных\_программ](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#%D0%9F%D1%80%D0%BE%D0%B8%D0%B7%D0%B2%D0%BE%D0%B4%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D1%81%D1%82%D1%8C_%D0%BE%D0%B1%D1%8A%D0%B5%D0%BA%D1%82%D0%BD%D1%8B%D1%85_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC), свободный - (дата обращения 25.11.2021)
3. Tiobe. TIOBE Index for November 2021 [Электронный ресурс]: the software quality company - / Tiobe. - Электронные данные. Режим доступа: URL.: <https://www.tiobe.com/tiobe-index/>, свободный - (дата обращения 25.11.2021)
4. Php.net. Руководство по PHP [Электронный ресурс]: php.net - / Php.net. - Электронные данные. Режим доступа: URL.: <https://www.php.net/manual/ru/>, свободный - (дата обращения 24.11.2021)
5. Кузнецов М., Симдянов И. Самоучитель PHP 7. — 2-е изд.. — СПб., 2018. — С. 448.
6. Котеров, Д.В. PHP5 / Д.В. Котеров, А.Ф. Костарев. - 2-е изд., перераб. и доп. - СПб.: БХВ-Петербург, 2011. - 1104 с.: ил. - (В подлиннике)

1. https://www.tiobe.com/tiobe-index/ [↑](#footnote-ref-1)