Лабораторная работа 4

Объектно-ориентированное программирование на PHP

Объектно-ориентированное программирование (ООП) – это методология (способ, подход) программирования согласно которой программный продукт представляется в виде взаимодействующих объектов.

Под *объектом* обычно понимается некая именованная сущность, хранящая в себе данные и имеющая своё собственное поведение. Поведение объекта и структуру данных, которые он хранит, а также способы доступа к этим данным, описывает класс, к которому этот объект принадлежит. *Класс* - это описание чем-то похожих, родственных объектов, которые обычно хранят одинаковый набор данных и имеют одинаковое поведение. Объект, который принадлежит какому-то конкретному классу, называют его *экземпляром.*

Три принципа ООП

ООП основано на трёх принципах, это:

- инкапсуляция
- полиморфизм
- наследование

Инкапсуляция – это принцип, согласно которому данные объединяются и хранятся в объектах, а также защищаются от неверного использования.

Наследование – это процесс приобретения одним типом объектов (классом) некоторых свойств другого типа объектов.

Полиморфизм – это использование одного и того же имени метода для решения нескольких похожих задач.

1 Классы и объекты в PHP

Класс - это базовое понятие в ООП. Классы образуют синтаксическую базу ООП. Их можно рассматривать как своего рода "контейнеры" для логически связанных данных и функций (обычно называемых методами).

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, то объект — как переменную (по аналогии). Скрипт может одновременно работать с несколькими объектами одного класса, как с несколькими переменными.

Описание классов в PHP начинаются служебным словом **class**. После него следует имя класса. Именовать классы принято с большой буквы. После имени класса в фигурных скобках следует описание членов класса – его полей (данных) и методов.

```
class Имя_класса {
// описание членов класса - свойств и методов для их обработки
}
```

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

```
Объект = new Имя_класса;
```

При описании полей (методов) класса нужно указывать спецификатор доступа – ключевое слово, которое будет определять область видимости поля, к которому оно относится. В php есть три спецификатора доступа: ***public, protected, private.*** Спецификатор ***public*** обеспечивает доступ к полю из любого места, ***protected*** **–** только из классов стоящих в той же цепочке наследования (из класса-потомка, из потомка потомка и т.д.), ***private*** запрещает доступ ото всюду, кроме самого класса. После спецификатора доступа идёт имя поля, предварённое знаком доллара.

Метод описывается так же, как и обыкновенная пользовательская функция. Методы также можно передавать параметры. Описания метода в классе, как и описание поля, начинается со спецификатора доступа, затем следует ключевое слово **function**, имя метода и список параметров в круглых скобках.

Пример:

```
Описание класса
  class  MyClass {

     // определение свойств
     // определение методов

  }

Инициализация класса (создание объекта)
  $myObj  =  new MyClass();
```

Доступ к полям класса

Для доступа к полям класса используется символ '->'. Имя поля, к которому мы хотим обратиться пишется без знака доллара. Значение полей класса по умолчанию можно установить прямо при объявлении любым литеральным (явно указанным) значением. При объявлении нельзя присвоить полю класса результат работы функции, за исключением array(), или экземпляр класса. Подобный приём считается дурным тоном, поскольку объявляя поля в классе, мы сообщаем, какие данные будут храниться в экземплярах этого класса. Для динамически определяемых свойств мы не можем гарантировать их наличие в экземпляре. Чтобы получить доступ к членам класса внутри класса, необходимо использовать указатель **$this**, который всегда относится к текущему объекту (которое, в отличие от многих других языков программирования, в PHP пишется со знаком доллара).

Пример:

```
Описание свойств

class MyClass {
      public $property1;
      protected $property2 = "value2";
      private $property3;
}

Доступ к свойствам класса

   $myObj = new MyClass();
   $myObj->property1;

Изменение значения свойств

   $myObj->property2 = "other_value";
```

Доступ к методам класса

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

Внутри метода доступ к текущему экземпляру класса можно получить при помощи ключевого слова $this. За пределами класса вызов методов производится с указанием имени экземпляра класса. Как и для доступа к полям, для вызова методов используется символ '->'.

Пример:

Описание методов

```
class MyClass {

    function myMethod($var1,$var2){
        // операторы
    }

}
```

Вызов метода

```
$myObj = new MyClass();
$myObj->myMethod('value1','value2');
```

Пример:

```
class MyClass {
  public $property1;

  function myMethod(){
      // Вывод значения свойства
      print($this->property1);
  }

  function callMethod(){
      // Вызов метода
      $this->myMethod();
  }
}
```

2 Конструкторы и Деструкторы

Для того, чтобы присвоить полям значения при создании экземпляра класса существуют конструкторы. Конструктор у класса в php может быть только один и если он не объявлен, то значения экземпляра остаются равными значениям по умолчанию. При объявлении конструктора указывается пишется ключевое слово ***function*** и ***__construct***. По умолчанию спецификатор доступа – public и его можно опустить. Конструктор, как и метод, имеет доступ ко всем полям класса через ключевое слово $this. Можно рассматривать конструктор как метод, который вызывается при создании экземпляра класса.

В старых версиях php имя конструктора должно было совпадать с именем класса, что на сегодняшний день является устаревшим синтаксисом.

Необходимость в вызове деструкторов возникает лишь при работе с объектами, использующими большой объем ресурсов, поскольку все переменные и объекты автоматически уничтожаются по завершении сценария. При объявлении деструктора указывается ключевое слово ***function*** и ***__construct***. Деструктор будет вызван при освобождении всех ссылок на определенный объект или при завершении скрипта. Деструктор будет вызван даже в том случае, если скрипт был остановлен с помощью функции exit(). Вызов exit() в деструкторе предотвратит запуск всех последующих функций завершения.

Пример:

```php
class MyClass {
   public $property;
   function __construct($var){
       $this->property = $var;
       echo "Вызван конструктор";
   }
   function __destruct(){
       echo "Вызван деструктор";
   }
}
//Вызов конструктора
$obj = new MyClass("Значение");
//Вызов деструктора
unset($obj);
```

Интересная подробность: в зависимости от количества передаваемых параметров могут вызываться разные конструкторы (например, отдельно для создания объекта и для инициализации полей).

3 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 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.

Клонирование объекта приводит к созданию новой копии объекта, при этом конструктор объекта не вызывается.

Приведем пример явного копирования и клонирования объекта с использованием метода **__clone()** .

Пример:

```
Явное копирование объектов
class MyClass {
  public $property;
}

$myObj = new MyClass();
$myObj->property = 1;
$myObj2 = clone $myObj;
$myObj2->property = 2;
print($myObj->property); // Печатает 1
print($myObj2->property); // Печатает 2
```

Пример:

```
class MyClass {
  public $property;
  function __clone(){
     $this->property = 2;
  }
}
$myObj = new MyClass();
$myObj->property = 1;
$myObj2 = clone $myObj;

print($myObj->property); // Печатает 1
print($myObj2->property); // Печатает 2
```

4 Наследование и полиморфизм 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, при этом исключая дублирование кода.

Унаследовать один класс от другого можно при помощи ключевого слова **extends**, которое можно перевести как «расширяет», что логично ведь класс-потомок, приобретая поля и методы своего класса-родителя, в то же время может содержать и свои собственные методы и поля, тем самым расширяя возможности своего класса-родителя.

Цепочка наследования может быть сколь угодно длинной и один класс может иметь сколь угодно много потомков, но предок у класса может быть только один, то есть в php не поддерживается множественное наследование.

Пример:

```php
class Car  {
   public $numWheels = 4;
   function printWheels() { echo $this->numWheels; }
}

class Toyota extends Car {
   public $country = 'Japan';
   function printCountry() { echo $this->country; }
}

$myCar = new Toyota();
$myCar->printWheels();
$myCar->printCountry();
```

В данном примере класс *Toyota* наследует класс *Car* и помимо свойства *$numWheels* и метода *printWheels* приобретает новое свойство – *$country* и метод – *printCountry.*

5 Перегрузка методов

Если в классе-потомке определить метод с именем, которое уже использовалось при объявлении другого метода в классе-предке, то такой метод будет называться переопределённым. Такой подход вполне себя оправдывает ведь классы, стоящие в одной цепочки наследования, описывают схожие множества объектов со схожим поведением, и для обозначения похожих действий таких объектов вполне естественно использовать одно и то же имя метода.

Пример:

```php
class Car  {
   public $numWheels = 4;
   function printWheels() { echo $this->numWheels; }
}
class Toyota extends Car {
   public $country = 'Japan';
   function printCountry() { echo $this->country; }
}
   function printWheels() {
       echo "Перегруженный метод printWheels() ";
   }
}

$myCar = new Toyota();
$myCar->printWheels();
```

В данном примере перегружен метод printWheels в дочернем классе Toyota.

Перегрузка свойств и методов классов

Обращения к свойствам объекта могут быть перегружены с использованием методов *__get* и *__set*. Эти методы будут срабатывать только в том случае, если объект или наследуемый объект не содержат свойства, к которому осуществляется доступ.

Вызовы методов могут быть перегружены с использованием метода *__call*. Этот метод будет срабатывать только в том случае, если объект или наследуемый объект не содержат метода, к которому осуществляется доступ.

6 Обращение к полям и методам класса-предка

Иногда бывает необходимо получить доступ к полю или методу класса-предка из класса-наследника. Использование методов, реализованных в классе-предке, иногда позволяет избавиться от дублирования кода. Например, обратите внимание на конструкторы классов User и SuperUser и заметьте, что в них дублируется код, присваивающий полям переданные в параметрах конструктора значения. Чтобы избежать подобного дублирования можно вызвать конструктор класса-предка из конструктора класса-потомка (в php в отличии от java или C# это не происходит автоматически)

Чтобы обратится к полям или методам класса-предка используется ключевое слово parent.

Пример:

```
class Car  {
   public $numWheels = 4;
   function printWheels() { echo $this->numWheels; }
}
class Toyota extends Car {
   public $country = 'Japan';
   function printWheels() {
       echo "Перегруженный метод printWheels() ";
       parent:: printWheels();
   }
}

$myCar = new Toyota();
$myCar->printWheels();
```

7 Обработка исключительных ситуаций (исключений)

Все пользователи Сети знают, как порой неприятно открывать страницу и видеть вместо долгожданной статьи/блога/картинки некий маловразумительный текст, сообщающий об ошибке сервера. К сожалению, реальность Интернет такова, что очень сложно сделать скрипт, одинаково хорошо работающий на любой платформе и при любых настройках сервера.

Поэтому очень важно уметь корректно выявлять и обрабатывать ошибки, которые могут возникнуть в скрипте. Для этого в PHP предусмотрено два механизма - обработка ошибок и обработка исключений.

С точки зрения разработчика, основное отличие ошибки от исключения в том, что после возникновения ошибки скрипт может продолжить выполнение, а после возникновения исключения - нет. Ещё одно различие - для обработки исключений необходимо использовать функции и специализированные языковые конструкции, а для обработки ошибок - только функции.

Исключения - это какие-либо аварийные ситуации, возникающие при выполнении скрипта. В PHP исключение можно сгенерировать ("выбросить", "вызвать") и поймать его. Исключение может сгенерироваться как интерпретатором, так и разработчиком.

Вызов исключения производится следующим образом:

```
<?php
	throw new Exception('My exception message');
?>
```

Перехват исключения осуществляется с помощью конструкции try...catch. В общем виде эта конструкция записывается так:

```
<?php
	try
	{
		// код, который может выбросить исключение
	}
	catch(Exception $ex)
	{
		//$ex - экземпляр класса Exception или его наследника
	}
?>
```

Стоит отметить, что блоков catch может быть много, по одному на каждый класс перехватываемых исключений. Таким образом можно создать *фильтр исключений*, т.е. перехватывать не все, а только избранные типы исключений, а все остальные будут перехвачены стандартным обработчиком PHP.

Пример:

```php
try {
   $a = 1;
   $b = 0;
   if($b == 0) throw new Exception("Деление на
   0!");
   echo $a/$b;
}catch(Exception $e){
   echo "Произошла ошибка - ",
   $e->getMessage(),     // Выводит сообщение
   " в строке ",
   $e->getLine(),  // Выводит номер строки
   " файла ",
   $e->getFile();  // Выводит имя файла
}
```

В данном примере код деления одного числа на другое заключен в блок *try*, чтобы перехватить и обработать ситуацию, если знаменатель (переменная $b) равен нулю. В этом случае генерируется новое исключение. Методы класса *Exception (getMessage, getLine, getFile)* позволяют получить детальную информацию о перехваченном исключении.

При необходимости можно создать собственный класс обработки исключений, унаследовав его от класса Exception. Собственный класс обработки исключений - это удобный инструмент разработчика, дающий возможность вести логи, отображать сообщения об ошибках, менять ход выполнения скрипта и много других возможностей.

Пример:

```php
class MathException extends Exception {
   function __construct($message) {
       parent::__construct($message);
   }
}
try {
   $a = 1;
   $b = 0;
   if ($b == 0) throw new MathException("Деление на
   0!");
   echo $a / $b;
} catch (MathException $e) {
   echo "Произошла математическая ошибка ",
       $e->getMessage(),
       " в строке ", $e->getLine(),
       " файла ", $e->getFile();
}
```

8 Константы класса

Константы могут быть объявлены в пределах одного класса. Значение должно быть неизменяемым выражением, не (к примеру) переменной, свойством, результатом математической операции или вызовом функции.

При объявлении константы не нужно указывать спецификатор доступа и обязательно указывать ключевое слово const. Имя константы указывается без знака доллара. Константы

принято именовать большими буквами, разделяя слова знаком подчёркивания. Константа так же доступна извне класса.

Пример:

```
class Human {
 const HANDS = 2;
 function printHands(){
      print (self::HANDS);// NOT $this!
 }
}

print ('Количество рук: '.Human::HANDS);
```

Создается константа HANDS. Показано, как обратиться к ней внутри класса и за его пределами.

9 Абстрактные методы и классы

PHP поддерживает определение абстрактных классов и методов. Абстрактные классы могут содержать описание абстрактных методов. Для таких методов указывается лишь заголовок с ключевым словом *abstract* и всеми прочими атрибутами, указываемыми при объявлении метода. Абстрактные методы не имеют тела или реализации, они лишь описывают, что должен уметь делать объект, а как он это будет делать – проблема наследников абстрактного класса. Класс, в котором объявлен хотя бы один абстрактный метод, должен также быть объявлен абстрактным.

В абстрактном классе можно объявлять так же и обычные методы и поля, которые могут быть унаследованы.

Экземпляр абстрактного класса создавать нельзя, так как в противном случае могла произойти попытка вызвать от этого экземпляра абстрактный метод, что абсурдно, так как он не имеет реализации.

Несмотря на то, что абстрактный класс не может иметь экземпляров, он может иметь конструктор, который могут использовать для инициализации его полей потомки.

Объявление абстрактного класса начинается с ключевого слова ***abstract.***

Пример:

```php
abstract class Car {
  public $petrol;
  function startEngine(){
        print('Двигатель завёлся!');
  }
  abstract function stopEngine();
}
class InjectorCar extends Car {
  public function stopEngine(){
       print('Двигатель остановился!');
  }
}
$myMegaCar = new Car();//Ошибка!
$myMegaCar = new InjectorCar();
$myMegaCar->startEngine();
$myMegaCar->stopEngine();
```

В примере создается абстрактный класс Car, имеющий обычное поле ($petrol) и метод (startEngine()), а также абстрактный метод (stopEngine()). Теперь класс, унаследованный от класса Car, обязан будет содержать реализацию метода stopEngine() или тоже должен быть объявлен как абстрактный, в противном случае еще до начала выполнения скрипта произойдет ошибка. В нашем случае дочерний класс InjectorCar содержит метод stopEngine() и описывает его действия.

10 Интерфейсы

Интерфейс, в отличие от абстрактного класса, не может содержать поля и методы, имеющие реализацию – он описывает только чистый функционал в виде абстрактных методов, которые должны реализовать его наследники.

Интерфейсы объявляются так же, как и обычные классы, но с использованием ключевого слова *"interface";* тела методов интерфейсов должны быть пустыми.

В отличие от абстрактных классов про интерфейсы чаще говорят, что классы их не наследуют, а имплементируют или реализуют. Если в классе, который реализует интерфейс, не реализованы все методы интерфейса, то он должен быть абстрактным.

Для включения интерфейса в класс программист должен использовать ключевое слово "*implements*" и описать функционал методов, перечисленных во включаемом интерфейсе.

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.

Стоит так же отметить ещё одно отличие интерфейсов от абстрактных классов – один класс может реализовывать сколь угодно много интерфейсов. Для этого их имена

просто нужно перечислить через запятую после ключевого слова implements. Унаследовать же несколько абстрактных классов нельзя. Это связанно с тем, что в абстрактных классах могут содержаться различные реализации не абстрактных методов с одинаковыми именами, и при наследовании этих классов не ясно, какую реализацию унаследовать потомку. В интерфейсах же реализованные методы отсутствуют и, если класс имплементирует несколько интерфейсов, в которых содержаться абстрактные методы с одинаковыми именами, то какой из этих методов будет реализован в классе безразлично.

В более широком смысле под интерфейсом часто понимают просто функционал, оторванный от реализации, то есть то, что что-то умеет делать и неважно как оно это делает.

Пример:

```php
interface Hand {
  function useKeyboard();
  function touchNose();
}

interface Foot {
  function runFast();
  function playFootball();
}

class Human implements Hand, Foot
{
  public function useKeyboard(){
        echo 'Use keyboard!';
  }
  public function touchNose(){
        echo 'Touch nose!';
  }
  public function runFast(){
        echo 'Run fast!';
  }
  public function playFootball(){
        echo 'Play football!';
  }
}
$vasyaPupkin = new Human();
$vasyaPupkin->touchNose();
```

В примере описаны два интерфейса (Hand, Foot) и методы, которые должны содержать наследники данных интерфейсов. Класс Human имплементирует оба интерфейса и уже содержит реализацию всех методов, заданных Интерфейсами.

Методы и классы final

Ключевое слово *final* позволяет помечать методы, чтобы наследующий класс не мог перегрузить их. После объявления класса *final* он не может быть унаследован.

11 Статические свойства и методы класса

В отличие от обычных, статические поля одинаковы для всех экземпляров класса и изменение статического свойства приведёт к его изменению для всех экземпляров класса. Обращение к статическим членам внутри класса производится при помощи ключевого слова *self*, а при их объявлении пишется ключевое слово *static*. Обращение к статическим

членам извне класса производится с указанием имени класса. Для доступа к ним используется символ '::'. Статические методы не имеют доступ к обычным (нестатическим) членам, так как в противном случае было бы непонятно к какому экземпляру класса относятся эти нестатические члены, к которым обращается статический метод.

Пример:

```
class CookieLover {
  static $loversCount = 0;
  function __construct(){
    ++self::$loversCount;
  }
  static function welcome(){
        echo 'Добро пожаловать в клуб любителей булочек!';
        //Никаких $this внутри статического метода!
  }
}
$vasyaPupkin = new CookieLover();
$frosyaBurlakova = new CookieLover();

print ('Текущее количество любителей булочек: '.
        CookieLover::$loversCount);
print (CookieLover::welcome());
```

В примере продемонстрирована работа со статическим полем $loverCount, обращение к нему как внутри класса, так и за его пределами. При создании нового экземпляра класса, значение статического поля увеличивается. Статический метод вывод сообщение – применение $this недопустимо.

12 Оператор instanceof

Этот оператор возвращает булево значение, показывающие относится ли объект к заданному классу или нет. Синтаксис этого оператора:

$object instanceof ClassName;

При этом этот оператор возвращает *true* даже если слева стоит экземпляр класса-наследника от класса, имя которого указанно справа.

Оператор *instanceof* удобно применять, когда необходимо убедится в наличии у экземпляра необходимых данных или функционала, которые присущи экземпляром какого-нибудь класса.

Пример:

```php
class Human {}
$myBoss = new Human();
if($myBoss instanceOf Human)
  print('Мой Босс – человек!');

class Woman extends Human {}
$englishQueen = new Woman();
if($englishQueen instanceOf Human)
  print('Английская королева – тоже человек!');

interface LotsOfMoney {}
class ReachPeople implements LotsOfMoney {}
$billGates = new ReachPeople();
if($billGates instanceOf LotsOfMoney)
  print('У Билла Гейтса много денег!');
```

13 Метод __toString()

Метод **__toString()** позволяет классу решать самостоятельно, как он должен реагировать при преобразовании в строку. Для того чтобы подставить значение переменных необходимо строку поместить в двойные кавычки.

Пример:

```php
class MyClass {
 function __toString(){
   return 'Вызван метод __toString()';
 }
}
$obj = new MyClass();
// Вызван метод __toString()
echo $obj;
```

14 Функция _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.

Начиная с 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. Это позволяет сделать автоматическую подгрузку файлов с описанием классов (в тот момент, когда мы начинаем использовать этот класс, в

случае, если класс в сценарии не используется- то и файл с его описанием не будет подключаться) и избавиться от множественных инструкций include.

Пример:

```
function __autoload($cl_name){
  print('Попытка создать объект класса
  '.$cl_name);
}
$obj = new undefinedClass();
```

Задания к лабораторной работе

Задание 1

1. Создайте класс User, в котором будут поля логин, пароль, email. Эти свойства должны заполняться при создании объекта этого класса.
2. Создайте методы доступа к свойствам класса.
3. Создайте метод, позволяющий вывести значения свойств объекта.
4. Создайте экземпляр класса User.
5. В классе User опишите метод __clone().Значения свойств по умолчанию: имя – "Guest", логин – "guest", пароль – "qwerty".
6. Клонируйте объект.
7. Создайте класс SuperUser – наследник класса User. Опишите свойство роль и создайте экземпляр класса SuperUser со свойством роль – админ. Распечатайте объект.
8. Сделайте все параметры конструктора класса User параметрами по умолчанию со значениями "пустая строка"("").
9. В конструкторе класса User генерируйте исключение, если введены не все данные.
10. Опишите перехват исключения и выводите в браузер сообщение об ошибке
11. Попробуйте создать экземпляр класса User без какого-либо параметра(-ов).
12. Создайте абстрактный класс AUser, объявите в нем абстрактный метод showInfo().
13. Обновите класс User, унаследовав его от абстрактного класса AUser. Внесите в класс User необходимые изменения. Запустите код и проверьте его работоспособность.

Задание 2

Создать класс-оболочку для работы с БД, минимальный функционал которого позволяет установить соединение с сервером и подключиться к БД, а также выполнить методы: получение данных, удаление данных, редактирование данных, очистка таблицы. Продемонстрируйте пример работы класса.

Задание 3

Создайте класс Session – оболочку над сессиями. Он должен иметь следующие методы: создать переменную сессии, получить переменную, удалить переменную сессии, проверить наличие переменной сессии. Сессия должна стартовать (session_start) в методе __construct. Продемонстрируйте пример работы класса.

Задание 4

Реализуйте класс Flash, который будет использовать внутри себя класс Session из предыдущей задачи.

Этот класс будет использоваться для сохранения сообщений в сессию и вывода их из сессии. *Зачем это нужно: такой класс часто используется для форм. Например на одной странице пользователь отправляет форму, мы сохраняем в сессию сообщение об успешной отправке, редиректим пользователя на другую страницу и там показываем сообщение из сессии*

Класс должен иметь два метода – setMessage, который сохраняет сообщение в сессию и getMessage, который получает сообщение из сессии.