Основы работы с ООП в PHP

Что такое объекты в жизни

Пусть есть автомобиль. У него есть колеса, цвет, вид кузова, объем двигателя и так далее. Кроме того, водитель может отдавать ему команды: ехать, остановится, повернуть направо, налево и тп.

Можно говорить о том, что существует некоторый *класс* автомобилей, обладающий общими свойствами (у всех есть колеса и всем им можно отдавать команды).

Конкретный автомобиль, стоящий на улице - это представитель этого класса, или, другими словами, *объект* этого класса. У всех объектов этого класса есть свойства (количество колес, цвет, вид кузова) и методы (ехать, остановится, повернуть направо, налево).

Если бы программировали автомобиль, то можно было бы сделать что-то вроде такого:

```
$мояМашина = new Автомобиль;

$мояМашина->едь();

$мояМашина->повернНалево();

$мояМашина->остановись();
```

Что в этом коде? Первой строчкой создаем объект $мояМашина - представителя класса Автомобиль. У нашей машины появляются свойства и методы, характерные для всех автомобилей - и мы можем отдавать ей команды. Можем также менять и свойства:

```
$мояМашина = new Автомобиль;

$мояМашина->цветКузова = 'Красный';

$мояМашина->количествоКолес = 10;
```

По сути, в реальной жизни всегда имеем дело с объектами - представителями определенного класса. Например, стол, за котором сидим, - это объект (конкретный представитель класса Стол). Как оказалось, программировать в терминах объектов и классов достаточно удобно.

Классы и объекты

Создать класс **User**, у которого будет два свойства - имя и возраст (:

```
<?php
	class User
	{
		public $name;
		public $age;
	}
?>
```

Пока наш класс ничего не делает, он просто описывает, что будут иметь объекты этого класса (в нашем случае каждый объект будет иметь имя и возраст). По сути,

пока мы не наплодим объектов нашего класса - он мертв и ничего не делает. Давайте же сделаем наш первый объект - представитель класса User:

```
<?php
	class User
	{
		public $name;
		public $age;
	}

	//Создадим объект нашего класса:
	$user = new User;
?>
```

Обратите внимание на то, что классы принято называть большими буквами (User), а объекты этих классов - маленькими ($user). Кроме того, можно писать **new User**, а можно - **new User()** - разницы нет.

Поменяем свойства нашего объекта, а потом выведем их на экран. Обращение к свойствам происходит через стрелочку "->" - *имяОбъекта->егоСвойство*:

```
<?php
	class User
	{
		public $name;
		public $age;
	}

	$user = new User;
	$user->name = 'Коля';
	$user->age = 25;

	echo $user->name; //выведет 'Коля'
	echo $user->age; //выведет 25
?>
```

В свойства объекта можно что-то записывать и из свойств можно выводить их содержимое. Создать 2 юзера (объекта) 'Коля' и 'Вася', заполнить их данными и вывести на экран сумму их возрастов:

```
<?php
	class User
	{
		public $name;
		public $age;
	}

	$user1 = new User;
	$user1->name = 'Коля';
	$user1->age = 25;

	$user2 = new User;
	$user2->name = 'Вася';
	$user2->age = 30;
```

```
        echo $user1->age + $user2->age;
?>
```

Методы

Методы - это функции, которые может вызывать каждый объект. При написании кода разница между методами и свойствами в том, что для методов надо писать круглые скобки в конце, а для свойств - не надо. Пример: $user->name - свойство, а $user->getName() - метод, который что-то делает.

Метод - это практически обычная функция, он может принимать параметры так же, как и все функции. Создать тренировочный метод ->show(), который будет выводить '!!!':

```
<?php
        class User
        {
                public $name;
                public $age;

                //Создаем метод:
                public function show()
                {
                        return '!!!';
                }
        }

        $user = new User;
        $user->name = 'Коля';
        $user->age = 25;

        //Вызовем наш метод:
        echo $user->show(); //выведет '!!!'
?>
```

Пусть метод ->show() выводит имя пользователя. Это имя можно получить в нашем методе - вопрос как. Если снаружи легко можем обратиться к имени через $user->name, то внутри класса следует делать так: **$this->name**.

Переменная **$this** указывает на объект класса и предназначена для использования внутри класса. Сделать так, чтобы метод show возвращал имя юзера:

```
<?php
        class User
        {
                public $name;
                public $age;

                //Создаем метод:
                public function show()
                {
                        //Имя доступно через $this:
                        return $this->name;
                }
        }
```

```
        $user = new User;
        $user->name = 'Коля';
        $user->age = 25;

        //Вызовем наш метод:
        echo $user->show(); //выведет 'Коля'
?>
```

С помощью переменной **$this** можно обратиться к свойствам объекта внутри самого кода объекта. Почему невозможно обратиться к ним, например, так: $user->name? Потому что переменной $user не существует в момент написания кода класса да и сам класс ничего не знает ни про какую переменную $user.

Через **$this** можно обращаться не только к свойствам объекта, но и к его методам. Сделать два метода - метод show(), который будет использовать внутри себя метод getName():

```
<?php
        class User
        {
                public $name;
                public $age;

                public function show()
                {
                        //Вызываем один метод внутри другого:
                        return $this->getName();
                }

                public function getName()
                {
                        return $this->name;
                }
        }

        $user = new User;
        $user->name = 'Коля';
        $user->age = 25;

        //Вызовем наш метод:
        echo $user->show(); //выведет 'Коля'
?>
```

Сделаем что-нибудь полезное с методами

Давайте расширим наш класс User - сделаем метод addYearsToAge, который будет добавлять заданное количество лет к возрасту:

```
<?php
        class User
        {
                public $name;
                public $age;

                public function addYearsToAge($years)
```

```
        {
                $this->age = $this->age + $years;
        }
    }

    $user = new User;
    $user->name = 'Коля';
    $user->age = 25;

    $user->addYearsToAge(3);
    echo $user->age; //выведет 28
?>
```

Разбираем public и private

Ключевое слово **public** указывает на то, что данные свойства и методы доступны извне (вне кода класса). В противоположность public есть ключевое слово **private**, которое указывает на то, что свойства и методы недоступны извне.
Зачем это надо? К примеру, есть класс, реализующий некоторый функционал. Есть набор методов, но часть этих методов является вспомогательными, и лучше, чтобы их нельзя было использовать вне класса - в этом случае легко можно подредактировать эти вспомогательные методы и можем быть уверенными в том, что их снаружи никто не использует и ничего страшного не случится.

Такой подход называется *инкапсуляцией* - все лишнее не должно быть доступно извне, в этом случае жизнь программиста станет проще.

То же самое касается и свойств. Некоторые свойства выполняют чисто вспомогательную функцию, и не должны быть доступны вне класса, иначе можно случайно подредактировать и сломать код.

ОБЪЯВИМ свойства $name и $age приватными и попытаемся обратиться к ним снаружи - сразу увидим ошибку PHP:

```
<?php
    class User
    {
        private $name;
        private $age;

    }

    $user = new User;
    //Выдаст ошибку, так как свойство name - private:
    $user->name = 'Коля';
?>
```

Геттеры и сеттеры

Очень часто все свойства объекта делают приватными, а для доступа к ним реализуют специальные методы.

Например, есть свойство **->name** - сделаем его приватным, а взамен сделаем метод *getName()*, который будет возвращать содержимое этого свойства во внешний мир, и метод *setName(новое имя)*, который будет менять значение этого свойства. Аналогично поступим с возрастом и вот, что у нас получится:

```
<?php
    class User
```

```
{
        private $name;
        private $age;

        public function getName()
        {
                return $this->name;
        }

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

        public function getAge()
        {
                return $this->age;
        }

        public function setAge($age)
        {
                $this->age = $age;
        }

}

$user = new User;
$user->setName('Коля');
echo $user->getName(); //выведет 'Коля'
?>
```

Во-первых, таким образом возможно создавать свойства только для чтения. Например, возраст можно будет только получить через getAge, но нельзя будет поменять без setAge.

Во-вторых, таким образом защищаемся от несанкционированных изменений этого свойства - его нельзя будет затереть случайно, а обязательно нужно будет вызвать метод.

Во-третьих, возможно проверять, к примеру, новые значения возраста - а вдруг кто-то пытается установить некорректный возраст - реализуем проверку возраста в setAge и будем изменять свойство age, только если возраст корректный. Такие методы, как мы реализовали, называются *геттерами* и *сеттерами*.

Рассмотрим private методы

Для сеттера setAge реализуем вспомогательный **private** метод checkAge, который будет проверять возраст на корректность перед его установкой:

```
<?php
        class User
        {
                private $name;
                private $age;
```

```
        public function getAge()
        {
                return $this->age;
        }

        public function setAge($age)
        {
                if ($this->checkAge($age)) {
                        $this->age = $age;
                }
        }

        private function checkAge($age)
        {
                if ($age < 100) {
                        return true;
                } else {
                        return true;
                }
        }

}

$user = new User;
$user->setAge(30);
echo $user->getAge();
?>
```

Метод __construct

Существует специальный встроенный метод **__construct**. Это метод (если мы его написали в коде) будет вызываться при создании объекта. Пример:

```
<?php
        class User
        {
                private $name;
                private $age;

                public function __construct()
                {
                        echo '!!!!';
                }
        }

        $user = new User; //выведет '!!!'
?>
```

В __construct можно также передавать параметры:

```
<?php
        class User
        {
                private $name;
```

```
        private $age;

        public function __construct($var)
        {
            echo $var;
        }
    }

    $user = new User('!!!'); //выведет '!!!'
?>
```

Чем может быть полезен этот метод? Давайте, для примера, сделаем объект User, для которого в момент создания объекта будут указываться имя и возраст. Эти значение в дальнейшем нельзя будет поменять, а только прочитать (геттеры сделаем, а сеттеры нет):

```
<?php
    class User
    {
        private $name;
        private $age;

        public function __construct($name, $age)
        {
            //Устанавливаем значения свойств:
            $this->name = $name;
            $this->age = $age;
        }

        public function getName()
        {
            return $this->name;
        }

        public function getAge()
        {
            return $this->age;
        }

    }

    $user = new User('Коля', 25);
    echo $user->getName();
    echo $user->getAge();
?>
```

Наследование

Представьте, что есть класс User. Он нужен для каких-то целей и в общем-то полностью устраивает - доработки этому классу в общем-то не нужны. Вот этот класс:

```
<?php
    class User
```

```
	{
		private $name;
		private $age;

		public function getName()
		{
			return $this->age;
		}

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

		public function getAge()
		{
			return $this->age;
		}

		public function setAge($age)
		{
			$this->age = $age;
		}

	}
?>
```

А теперь представим себе ситуацию, когда нам понадобился еще и класс Worker (работник). Работник очень похож на юзера, имеет те же свойства и методы, но еще и добавляет свои - свойство $salary (зарплата), а также геттеры и сеттеры для этого свойства.

Что же делать в этом случае? Глупо копировать код класса User и вставлять его в Worker - дублирование кода мы не любим.

Лучше всего будет *унаследовать* класс Worker от класса User - в этом случае у работника будут все свойства и методы родительского класса (кроме private!) и он спокойно сможет добавить к ним свои.

Наследование реализуется с помощью ключевого слова **extends**, вот так: *class Worker extends User*. Вот реализация класса Worker:

```
<?php
	class Worker extends User
	{
		private $salary;

		public function getSalary()
		{
			return $this->salary;
		}

		public function setSalary($salary)
		{
			$this->salary = $salary;
		}
```

```
        }

        $worker = new Worker();
        $worker->setSalary(1000);
        $worker->setName('Коля'); //метод родителя
        $worker->setAge(25); //метод родителя

        echo $worker->getSalary();
        echo $worker->getName(); //метод родителя
        echo $worker->getAge(); //метод родителя
?>
```

Пусть кроме работника нужен еще и студент **Student** - унаследуем его от **User**:

```
<?php
        class Student extends User
        {
                private $course; //курс

                public function getCourse()
                {
                        return $this->course;
                }

                public function setCourse($course)
                {
                        $this->course = $course;
                }

        }

        $student = new Student();
        $student->setName('Коля');
        $student->setAge(25);
        $student->setCourse(3);

        echo $student->getName();
        echo $student->getAge();
        echo $student->getCourse();
?>
```

Ключевое слово protected

private свойства и методы не наследуются. Это не мешает работать public геттерам и сеттерам, унаследованным от User, например: *$student->setName()* работает, но напрямую получить свойство name внутри класса потомка не сможем - это приведет к ошибке. Нужные свойства и методы можно объявить как **protected** - в этом случае они станут доступны в потомках, но по-прежнему не будут доступны извне. В классе Student реализуем метод **addOneYear** - он будет добавлять 1 год к свойству age. Однако, если в классе User свойство age оставить приватным - увидим ошибку:

```
<?php
```

```
        class Student extends User
        {
                private $course;

                public function getCourse()
                {
                        return $this->course;
                }

                public function setCourse($course)
                {
                        $this->course = $course;
                }

                //Реализуем этот метод:
                public function addOneYear()
                {
                        $this->age++; //выдаст ошибку
                }

        }

        $student = new Student();
        $student->setName('Коля');
        $student->setAge(25);
        $student->setCourse(3);

        echo $student->addOneYear();
        echo $student->getAge();
?>
```

Для исправления ошибки поправим класс User - сделаем его свойства **protected**, а не private:

```
<?php
        class User
        {
                protected $name;
                protected $age;
                ...
        }
?>
```

Перегрузка и parent::

Пусть есть класс Student, наследующий от класса User метод setAge. Предположим, что этот метод setAge от родителя нам чем-то не подходит и мы хотим написать свою реализацию в классе-потомке. Так можно делать (это называется *перегрузка*).

Давайте напишем студент свой setAge в классе Student. Наш setAge будет проверять то, что возраст студента меньше 25 лет:

```
<?php
```

```
	class Student extends User
	{
		private $course;

		//Перезаписываем метод родителя:
		public function setAge($age)
		{
			if ($age <= 25) {
				$this->age = $age;
			}
		}
	}

	$student = new Student();
	$student->setAge(25);

	echo $student->getAge();
?>
```

Пусть мы затерли (*перегрузили*) метод родителя, но хотели бы использовать и его. То есть мы хотим иметь в классе Student свой метод setAge а также метод родителя setAge. В этом случае к методу родителя можно обратиться так: **parent::setAge()**.

Давайте доработаем наш класс Student так, чтобы использовался метод **setAge родителя** (убираем некоторое дублирование кода таким образом):

```
<?php
	class Student extends User
	{
		private $course;

		public function setAge($age)
		{
			if ($age <= 25) {
				parent::setAge($age);
			}
		}
	}

	$student = new Student();
	$student->setAge(25);

	echo $student->getAge();
?>
```

На использование классов внутри других классов

Бывает такое, что мы хотели бы использовать методы одного класса внутри другого, но не хотели бы наследовать от этого класса. Пусть есть класс User, который хочет использовать класс работы с базами данных Db. Создадим внутри User новый объект класса Db, запишем его в любую переменную, например, в **$this->db** и будем спокойно использовать public методы и свойства класса Db:

```
<?php
```

```
	class User
	{
		private $db;

		public function __construct($id)
		{
			//Создаем объект для работы с БД:
			$this->db = new Db;
		}
	}
?>
```

Некоторая практика

Напишем реализацию класса Db и класса User. Попробуйте сами разобраться в этом коде:

```
<?php
	class Db
	{
		private $link;
		private $host = 'localhost';
		private $user = 'root';
		private $password = '';
		private $database = 'test';
		private $table = 'users';

		//Подключается к базе:
		public function __construct()
		{
			$this->link = mysqli_connect(...);
		}

		//Делает запрос к базе:
		public function get($id, $field)
		{
			$query = $this->createSelect($id, $field);
			$result = $this->makeQuery($query); //будет в виде
['age'=>25]
			return $result[$field]; //а тут достанем 25
		}

		//Создает строку с запросом:
		private function createSelect($id, $field)
		{
			$table = $this->table;
			return "SELECT $field FROM $table WHERE id=$id";
		}

		//Совершает запрос к базе:
		private function makeQuery($query)
		{
```

```
                $result = mysqli_query($this->link, $query);
                return mysqli_fetch_assoc($result);
            }
        }
?>
```

Класс User:

```
<?php
        class User
        {
                private $id;
                private $db;

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

                        //Создаем объект для работы с БД:
                        $this->db = new Db;
                }

                public function getName()
                {
                        return $this->db->get($this->id, 'name');
                }

                public function getAge()
                {
                        return $this->db->get($this->id, 'age');
                }

        }
?>
```

Переменные названия свойств и методов

Названия свойств и методов можно хранить в переменной. К примеру, есть переменная **$var**, в которой лежит строка 'name'. Тогда обращение **$user->$var** по сути эквивалентно обращению **$user->name**.

Задачи для решения

Работа с классами и объектами

1. Сделайте класс **Worker**, в котором будут следующие **public** поля - name (имя), age (возраст), salary (зарплата).

Создайте объект этого класса, **затем** установите поля в следующие значения (не в __construct, а для созданного объекта) - имя 'Иван', возраст 25, зарплата 1000. Создайте второй объект этого класса, установите поля в следующие значения - имя 'Вася', возраст 26, зарплата 2000.

Выведите на экран сумму зарплат Ивана и Васи. Выведите на экран сумму возрастов Ивана и Васи.

2. Сделайте класс **Worker**, в котором будут следующие **private** поля - name (имя), age (возраст), salary (зарплата) и следующие **public** методы setName, getName, setAge, getAge, setSalary, getSalary.

Создайте 2 объекта этого класса: 'Иван', возраст 25, зарплата 1000 и 'Вася', возраст 26, зарплата 2000.

Выведите на экран сумму зарплат Ивана и Васи. Выведите на экран сумму возрастов Ивана и Васи.

3. Дополните класс **Worker** из предыдущей задачи **private** методом checkAge, который будет проверять возраст на корректность (от 1 до 100 лет). Этот метод должен использовать метод setAge перед установкой нового возраста (если возраст не корректный - он не должен меняться).

На __construct

4. Сделайте класс **Worker**, в котором будут следующие **private** поля - name (имя), salary (зарплата). Сделайте так, чтобы эти свойства заполнялись в методе __construct при создании объекта (вот так: new Worker(имя, возраст)). Сделайте также **public** методы getName, getSalary.

Создайте объект этого класса 'Дима', возраст 25, зарплата 1000. Выведите на экран произведение его возраста и зарплаты.

Наследование

5. Сделайте класс **User**, в котором будут следующие **protected** поля - name (имя), age (возраст), **public** методы setName, getName, setAge, getAge.

Сделайте класс **Worker**, который наследует от класса User и вносит дополнительное **private** поле salary (зарплата), а также методы **public** getSalary и setSalary.

Создайте объект этого класса 'Иван', возраст 25, зарплата 1000. Создайте второй объект этого класса 'Вася', возраст 26, зарплата 2000. Найдите сумму зарплата Ивана и Васи.

Сделайте класс **Student**, который наследует от класса User и вносит дополнительные **private** поля стипендия, курс, а также геттеры и сеттеры для них.

6. Сделайте класс **Driver** (Водитель), который будет наследоваться от класса Worker из предыдущей задачи. Этот метод должен вносить следующие private поля: водительский стаж, категория вождения (A, B, C).

Практика

7. Создайте класс **Form** - оболочку для создания форм. Он должен иметь методы input, submit, password, textarea, open, close. Каждый метод принимает массив атрибутов.

Примеры использования:

```php
<?php
        echo $form->input(['type'=>'text', 'value'=>'!!!']);
        //Код выше выведет <input type="text" value="!!!">

        echo $form->password(['value'=>'!!!']);
        //Код выше выведет <input type="password" value="!!!">

        echo $form->submit(['value'=>'go']);
        //Код выше выведет <input type="submit" value="go">

        echo $form->textarea(['placeholder'=>'123', 'value'=>'!!!']);
        //Код выше выведет <textarea placeholder="123">!!!</textarea>

        echo $form->open(['action'=>'index.php', 'method'=>'POST']);
        //Код выше выведет <form action="index.php" method="POST">

        echo $form->close();
        //Код выше выведет </form>
?>
```

Передаваемые атрибуты могут быть любыми:

```php
<?php
        echo $form->input(['type'=>'text', 'value'=>'!!!', 'class'=>'ggg']);
        //Код выше выведет <input type="text" value="!!!" class="ggg">
?>
```

Для решения задачи сделайте **private** метод, который параметром будет принимать массив, например, *['type'=>'text', 'value'=>'!!!']* и делать из него строку с атрибутами, в нашем случае *type="text" value="!!!"*.

Пример создания формы с помощью нашего класса:

```php
<?php
        echo $form->open(['action'=>'index.php', 'method'=>'POST']);
        echo $form->input(['type'=>'text', 'placeholder'=>'Ваше имя', 'name'=>'name']
);
        echo $form->password(['placeholder'=>'Ваш пароль', 'name'=>'pass']);
        echo $form->submit(['value'=>'Отправить']);
        echo $form->close();
?>
```

В результате получится следующая форма:

```html
<form action="index.php" method="POST">

        <input type="text" placeholder="Ваше имя" name="name">

        <input type="text" placeholder="Ваш пароль" name="pass">

</form>
```

8. Создайте класс **SmartForm**, который будет наследовать от **Form** из предыдущей задачи и сохранять значения инпутов и textarea после отправки.

То есть если мы сделали форму, нажали на кнопку отправки - то значения из инпутов не должны пропасть. Мало ли что-то пойдет не так, например, форма некорректно заполнена, а введенные данные из нее пропали и их следует вводить заново. Этого следует избегать.

9. Создайте класс **Cookie** - оболочку над работой с куками. Класс должен иметь следующие методы: установка куки *set(имя куки, ее значение)*, получение куки *get(имя куки)*, удаление куки *del(имя куки)*.

10. 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**.

11. Реализуйте класс **Flash**, который будет использовать внутри себя класс Session из предыдущей задачи (именно использовать, а не наследовать).

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 да метода - **setMessage**, который сохраняет сообщение в сессию и **getMessage**, который получает сообщение из сессии.

12. Создайте класс-оболочку **Db** над базами данных. Класс должен иметь следующие методы: получение данных, удаление данных, редактирование данных, подсчет данных, очистка таблицы, очистка таблиц.

13. Создайте класс **Log** для ведения логов. Этот класс должен иметь следующие методы: сохранить в лог, получить последние N записей, очистить таблицу с логами.

Класс **Log** должен использовать класс **Db** из предыдущей задачи (именно использовать, а не наследовать).