Алгоритмизация и программирование

Лекция 7

Введение в классы и объекты в PHP

С этой лекции мы с вами начинаем изучение ООП в PHP.

Давайте рассмотрим пример из жизни, а потом перенесем его на PHP.

В качестве примера возьмем 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.

Другими словами, сам класс — это некий проект, по которому на заводе делаются автомобили. Объект же — это сама машина, сделанная по чертежам данного проекта.

В PHP класс создается с помощью ключевого слова class, за которым следует название этого класса. Давайте сделаем класс Car:

```php
<?php
class Car
{
    // тут код, то есть PHP-чертеж автомобиля
}
?>
```

Укажем теперь в нашем чертеже, что любой автомобиль, созданный по нашему проекту, будет иметь свойство для цвета и свойство для количества топлива.

Для этого внутри класса напишем свойство $color и свойство $fuel:

```php
<?php
class Car
{
    // Зададим свойства (по сути переменные класса):
    public $color; // цвет автомобиля
    public $fuel; // количество топлива
}
?>
```

Давайте теперь сделаем методы нашего класса. В PHP методы, подобно обычным функциям, объявляются с помощью ключевого слова function, перед которым пишется ключевое слово public.

Как уже упоминалось выше, наш автомобиль может ехать, может поворачивать, может останавливаться. Сделаем соответствующие методы в нашем классе:

```php
<?php
class Car
{
    public $color; // цвет автомобиля
    public $fuel; // количество топлива
    // Команда ехать:
    public function go()
    {
        // какой-то PHP код
    }
    // Команда поворачивать:
    public function turn()
    {
        // какой-то PHP код
    }
    // Команда остановиться:
    public function stop()
    {
        // какой-то PHP код
    }
}
?>
```

Мы с вами сделали чертеж нашего автомобиля. Теперь нужно отправится на завод и сделать объект этого класса (то есть конкретный автомобиль).

В PHP это делается с помощью ключевого слова new, после которого пишется имя класса:

```
<?php
new Car; // командуем заводу сделать автомобиль
?>
```

Однако, если просто создать объект класса - это ни к чему не приведет (это все равно, что, к примеру, объявить массив и ничего в него не записать). Нам нужна переменная для хранения этого объекта.

Пусть эта переменная будет называться $myCar - запишем в нее созданный нами объект:

```
<?php
$myCar = new Car; // запишем созданный объект в переменную $myCar
?>
```

После создания автомобиля можно обращаться к его свойствам. Обращение к ним происходит через стрелочку ->. Давайте установим свойства нашего объекта:

```
<?php
$myCar = new Car; // командуем заводу сделать автомобиль
// Устанавливаем свойства объекта:
$myCar->color = 'red'; // красим в красный цвет
$myCar->fuel = 50; // заливаем топливо
?>
```

Все, наш автомобиль создан, покрашен и заправлен. Теперь мы можем отдавать ему команды через методы этого автомобиля.

Обращение к методам также происходит через стрелочку, но, в отличие от свойства, после имени метода следует писать круглые скобки. Давайте покомандуем нашим объектом:

```
<?php
$myCar->go();   // автомодиль->едь
$myCar->turn(); // автомодиль->поверни
$myCar->stop(); // автомодиль->остановись
?>
```

Работа со свойствами объектов на PHP

Сейчас мы с вами научимся работать с объектами и их свойствами на более практическом примере. Давайте создадим класс User, который будет описывать юзера нашего сайта.

Пусть у нашего пользователя будет два свойства: имя и возраст. Напишем код нашего класса:

```php
<?php
class User
{
    public $name; // свойство для имени
    public $age; // свойство для возраста
}
?>
```

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 создадим хотя бы один объект нашего класса - ничего полезного не произойдет.

Давайте создадим объект нашего класса. При этом нужно иметь ввиду, что классы принято называть большими буквами, а объекты этих классов - маленькими:

```php
<?php
// Объявляем класс:
class User
{
    public $name;
    public $age;
}
// Создаем объект нашего класса:
$user = new User;
?>
```

Давайте теперь что-нибудь запишем в свойства нашего объекта, а потом выведем эти данные на экран:

```php
<?php
class User
{
    public $name;
    public $age;
}
$user = new User; // создаем объект нашего класса
$user->name = 'john'; // записываем имя в свойство name
$user->age = 25; // записываем возраст в свойство age
echo $user->name; // выводим записанное имя
echo $user->age; // выводим записанный возраст
?>
```

Как вы уже поняли - в свойства объекта можно что-то записывать и из свойств можно выводить их содержимое. Давайте теперь сделаем 2 объекта-юзера: 'john' и 'eric', заполним их данными и выведем на экран сумму их возрастов:

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

```
$user1 = new User; // создаем первый объект
$user1->name = 'john'; // записываем имя
$user1->age = 25; // записываем возраст
// Второй объект
$user2 = new User; // создаем второй объект
$user2->name = 'eric'; // записываем имя
$user2->age = 30; // записываем возраст
// Найдем сумму возрастов:
echo $user1->age + $user2->age; // выведет 55
?>
```

Работа с методами объектов

Перейдем теперь к методам. Методы - это по сути 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.

Давайте потренируемся - сделаем метод show(), который будет выводить некоторый текст:

```
<?php
class User
{
    public $name;
    public $age;
    // Создаем метод:
    public function show()
    {
        return '!!!';
    }
}
$user = new User;
$user->name = 'john';
$user->age = 25;
// Вызовем наш метод:
echo $user->show(); // выведет '!!!'
?>
```

Параметры метода

Так как метод — это, по сути, обычная функция, он может принимать параметры так же, как и все функции.

Давайте сделаем так, чтобы наш метод show() параметром принимал какую-нибудь строку и добавлял ей в конец '!!!':

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

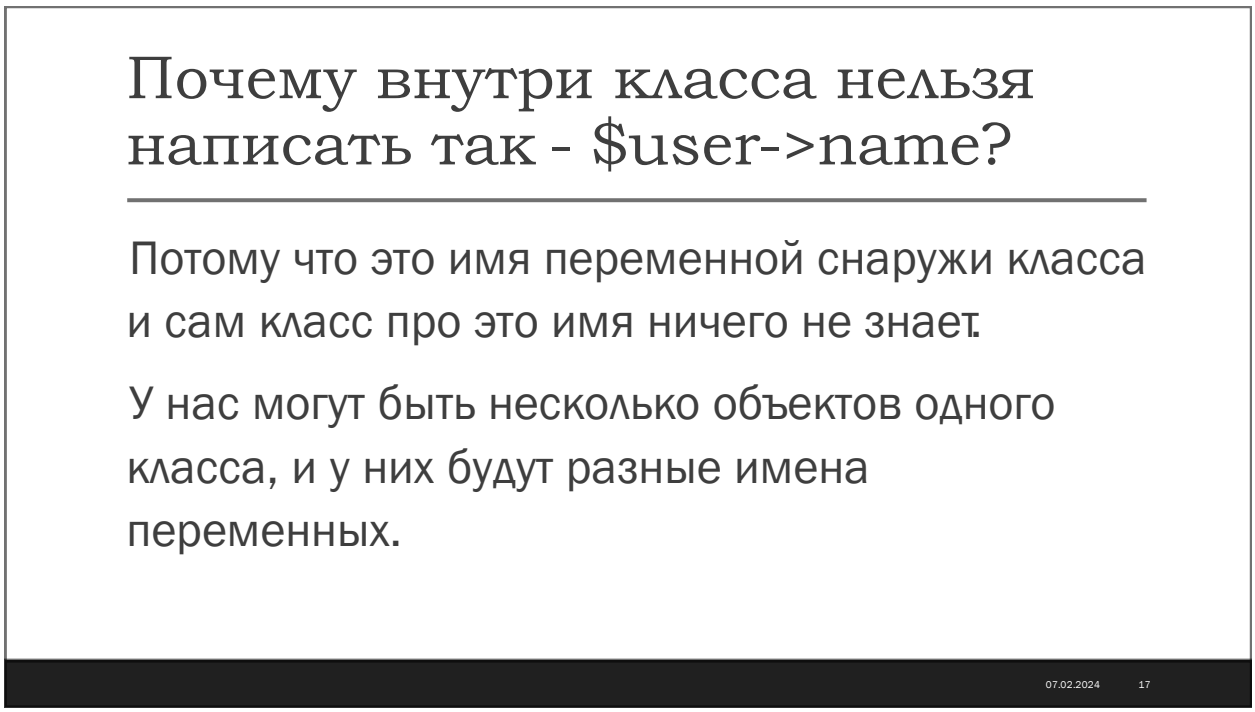
```
    // Создаем метод:
    public function show($str)
    {
        return $str . '!!!';
    }
}
$user = new User;
$user->name = 'john';
$user->age = 25;
// Вызовем наш метод:
echo $user->show('hello'); // выведет 'hello!!!'
?>
```

Обращение к свойствам класса через $this

Пусть теперь наш метод show() выводит нечто полезное - имя пользователя, которое хранится в свойстве name. Для того, чтобы обратиться к свойству класса внутри метода этого класса, вместо имени объекта следует писать специальную переменную $this:

```
<?php
class User
{
    public $name;
    public $age;
    public function show()
    {
        return $this->name; // вернем имя из свойства
    }
}
?>
```

Почему внутри класса нельзя написать так - $user->name?



Потому что это имя переменной снаружи класса и сам класс про это имя ничего не знает (более того, у нас же могут быть несколько объектов одного класса, и у них будут разные имена переменных).

Давайте создадим объект нашего класса и проверим работу метода show():

```
<?php
class User
{
    public $name;
    public $age;
    public function show()
    {
        // Возвращаем имя:
        return $this->name;
    }
}
$user = new User; // создаем объект класса
$user->name = 'john'; // записываем имя
$user->age = 25; // записываем возраст
// Вызываем наш метод:
echo $user->show(); // выведет 'john'
?>
```

Запись свойств

С помощью $this свойства можно не только прочитывать, но и записывать. Давайте сделаем метод setName(), который параметром будем принимать имя пользователя и записывать его в свойство name:

```
<?php
```

```
class User
{
    public $name;
    public $age;
    // Метод для изменения имени юзера:
    public function setName($name)
    {
        $this->name = $name; // запишем новое значение свойства name
    }
}
$user = new User; // создаем объект класса
$user->name = 'john'; // записываем имя
$user->age = 25; // записываем возраст
// Установим новое имя:
$user->setName('eric');
// Проверим, что имя изменилось:
echo $user->name; // выведет 'eric'
?>
```

Обращение к методам класса через $this

Через $this можно обращаться не только к свойствам объекта, но и к его методам. Посмотрим на примере. Пусть у нас есть класс User, а в нем метод setAge для изменения возраста юзера:

```
<?php
class User
{
    public $name;
    public $age;
    // Метод для изменения возраста юзера:
    public function setAge($age)
    {
        $this->age = $age;
    }
}
?>
```

Давайте добавим проверку введенного возраста: если он от 18 до 60, то будем менять возраст на новый, а если нет - то менять не будем:

```
<?php
class User
{
    public $name;
    public $age;
    // Метод для изменения возраста юзера:
    public function setAge($age)
    {
        // Если возраст от 18 до 60:
        if ($age >= 18 and $age <= 60) {
            $this->age = $age;
        }
    }
}
?>
```

Пусть также нам нужен метод addAge, который будет добавлять некоторое количество лет к текущему возрасту:

```
<?php
class User
{
    public $name;
    public $age;
    // Метод для изменения возраста юзера:
    public function setAge($age)
    {
        // Если возраст от 18 до 60:
        if ($age >= 18 and $age <= 60) {
            $this->age = $age;
        }
    }
    // Метод для добавления к возрасту:
    public function addAge($years)
    {
        $this->age = $this->age + $years;
    }
}
?>
```

В метод addAge, конечно же, также необходимо добавить проверку возраста, сделаем это:

```
<?php
class User
{
    public $name;
    public $age;
    // Метод для изменения возраста юзера:
    public function setAge($age)
    {
        // Если возраст от 18 до 60:
        if ($age >= 18 and $age <= 60) {
            $this->age = $age;
        }
    }
    // Метод для добавления к возрасту:
    public function addAge($years)
    {
        $newAge = $this->age + $years; // вычислим новый возраст
        // Если НОВЫЙ возраст от 18 до 60:
        if ($newAge >= 18 and $newAge <= 60) {
            $this->age = $newAge; // обновим, если новый возраст
прошел проверку
        }
    }
}
?>
```

Получится, что ограничения на возраст теперь задаются в двух местах (в методе setAge и в методе addAge), что не очень хорошо: если мы захотим

поменять ограничение, нам придется сделать это в двух местах - это неудобно, и в каком-то из мест мы можем забыть внести изменения - это опасно, ведь наш код получится с трудноуловимой ошибкой.

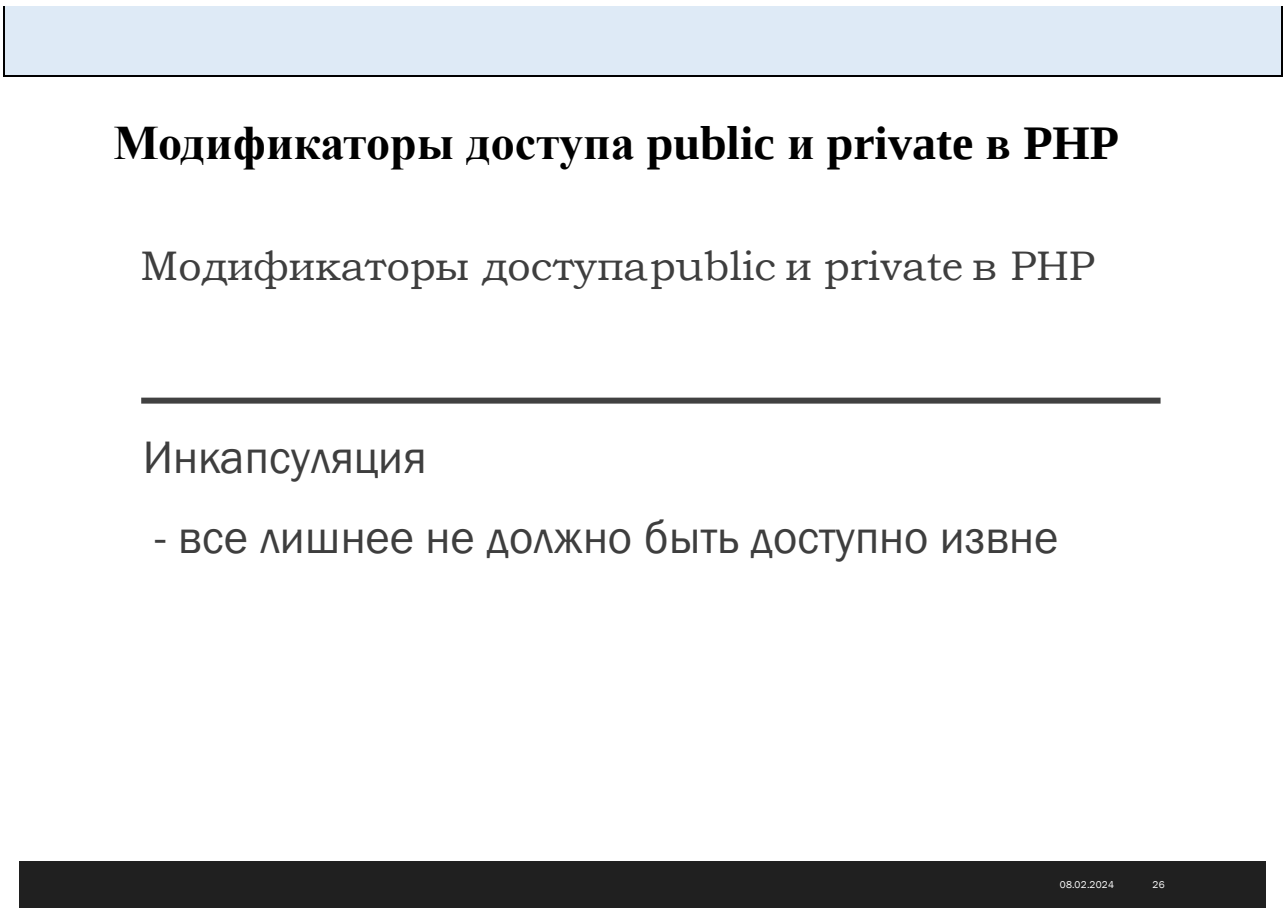
Давайте вынесем проверку возраста на корректность в отдельный вспомогательный метод isAgeCorrect, в который параметром будет передаваться возраст для проверки, и используем его внутри методов setAge и addAge:

```
<?php
class User
{
    public $name;
    public $age;
    // Метод для проверки возраста:
    public function isAgeCorrect($age)
    {
        if ($age >= 18 and $age <= 60) {
            return true;
        } else {
            return false;
        }
    }
    // Метод для изменения возраста юзера:
    public function setAge($age)
    {
        // Проверим возраст на корректность:
        if ($this->isAgeCorrect($age)) {
            $this->age = $age;
        }
    }
    // Метод для добавления к возрасту:
    public function addAge($years)
    {
        $newAge = $this->age + $years; // вычислим новый возраст
        // Проверим возраст на корректность:
        if ($this->isAgeCorrect($newAge)) {
            $this->age = $newAge; // обновим, если новый возраст
прошел проверку
        }
    }
}
?>
```

Теперь любое изменения в условиях проверки можно будет легко сделать в одном месте. Проверим, что все работает как надо:

```
<?php
$user = new User;
$user->setAge(30); // установим возраст в 30
echo $user->age; // выведет 30
$user->setAge(10); // установим некорректный возраст
echo $user->age; // не выведет 10, а выведет 30
?>
```

Модификаторы доступа public и private в PHP



Ключевое слово public, которое мы пишем перед именами указывает на то, что данные свойства и методы доступны извне (вне кода класса). В противоположность ключевое слово private, которое указывает на то, что свойства и методы недоступны извне.

Зачем это надо? К примеру, у нас есть класс, реализующий некоторый функционал. Есть набор методов, но часть этих методов является вспомогательными.

Будет лучше, чтобы эти вспомогательные методы нельзя было использовать вне нашего класса. В этом случае мы легко сможем редактировать код этих вспомогательных методов и будем уверенны 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 мы их можем случайно поменять снаружи и сломать работу нашего кода.

Методы и свойства, которые мы хотим сделать недоступными извне, называются приватными и объявляются с помощью ключевого слова private.

Давайте попробуем - объявим свойства $name и $age приватными и попытаемся обратиться к ним снаружи - мы сразу увидим ошибку:

```php
<?php
class User
{
    private $name;
    private $age;
}
$user = new User;
// Выдаст ошибку, так как свойство name - private:
$user->name = 'john';
?>
```

Применим на практике

Давайте посмотрим на класс User, который мы сделали в предыдущем уроке:

```php
<?php
class User
{
    public $name;
    public $age;
    // Метод для проверки возраста:
    public function isAgeCorrect($age)
    {
        return $age >= 18 and $age <= 60;
    }
    // Метод для изменения возраста юзера:
    public function setAge($age)
    {
        // Проверим возраст на корректность:
        if ($this->isAgeCorrect($age)) {
            $this->age = $age;
        }
    }
    // Метод для добавления к возрасту:
    public function addAge($years)
    {
        $newAge = $this->age + $years; // вычислим новый возраст
        // Проверим возраст на корректность:
        if ($this->isAgeCorrect($newAge)) {
            $this->age = $newAge; // обновим, если новый возраст
прошел проверку
        }
    }
}
?>
```

Как мы знаем, метод isAgeCorrect является вспомогательным и мы не планируем использовать его снаружи класса. Логично сделать его приватным, чтобы другой программист, который будет потом работать над нашим проектом (или мы сами через некоторое время) случайно не использовал этот метод снаружи класса.

Сделаем это:

```php
<?php
class User
{
    public $name;
    public $age;
    // Объявим приватным:
    private function isAgeCorrect($age)
    {
        return $age >= 18 and $age <= 60;
    }
    // Метод для изменения возраста юзера:
    public function setAge($age)
    {
        // Проверим возраст на корректность:
        if ($this->isAgeCorrect($age)) {
            $this->age = $age;
        }
    }
    // Метод для добавления к возрасту:
    public function addAge($years)
    {
        $newAge = $this->age + $years; // вычислим новый возраст
        // Проверим возраст на корректность:
        if ($this->isAgeCorrect($newAge)) {
            $this->age = $newAge; // обновим, если новый возраст
прошел проверку
        }
    }
}
?>
```

Обычно все приватные методы размещают в конце класса, давайте перенесем наш метод в самый низ:

```php
<?php
class User
{
    public $name;
    public $age;
    // Метод для изменения возраста юзера:
    public function setAge($age)
    {
        // Проверим возраст на корректность:
        if ($this->isAgeCorrect($age)) {
            $this->age = $age;
        }
    }
    // Метод для добавления к возрасту:
    public function addAge($years)
    {
        $newAge = $this->age + $years; // вычислим новый возраст
        // Проверим возраст на корректность:
        if ($this->isAgeCorrect($newAge)) {
            $this->age = $newAge; // обновим, если новый возраст
прошел проверку
        }
    }
```

```
    // Метод для проверки возраста:
    private function isAgeCorrect($age)
    {
        return $age >= 18 and $age <= 60;
    }
}
?>
```

Существует специальное правило: если вы делаете новый метод и не знаете, сделать его публичным или приватным, - делайте приватным. В дальнейшем, если он понадобится снаружи, - вы поменяете его на публичный.

Еще раз резюмируем: слова public и private не нужны для реализации логики программы, а нужны для того, чтобы уберечь программистов от ошибок.

Конструктор объекта в ООП на PHP

Рассмотрим следующий код:

```
<?php
// Класс с публичными свойствами name и age:
class User
{
    public $name;
    public $age;
}
// Создаем объект класса:
$user = new User;
// Записываем данные:
$user->name = 'john';
$user->age = 25;
// Прочитываем данные:
echo $user->name; // выведет 'john'
echo $user->age; // выведет 25
?>
```

В данном коде не очень удобно то, что легко можно забыть записать данные в какое-нибудь свойство объекта, особенно если этих свойств много.

Было бы удобно этот код:

```
<?php
// Создаем объект класса:
$user = new User;
// Записываем данные:
$user->name = 'john';
$user->age = 25;
?>
```

Заменить на вот этот:

```
<?php
$user = new User('john', 25); // создадим объект, сразу заполнив его
данными
```

```
?>
```

То есть сделать так, чтобы поля объекта заполнялись при его создании - в этом случае мы никак не сможем забыть задать значения этих полей.

Для решения проблемы нам поможет метод конструктор с названием __construct. Суть в следующем - если в коде класса существует метод с таким названием - он будет вызываться в момент создания объекта:

```
<?php
class User
{
    public $name;
    public $age;
    // Конструктор объекта:
    public function __construct()
    {
        echo '!!!';
    }
}
$user = new User; // выведет '!!!'
?>
```

Конструктор в общем-то такой же метод, как и все остальные и может принимать параметры, как на этом примере:

```
<?php
class User
{
    public $name;
    public $age;
    public function __construct($var1, $var2)
    {
        echo $var1 + $var2; // найдем сумму параметров
    }
}
$user = new User(1, 2); // выведет 3
?>
```

Итак, давайте переделаем наш код, применив конструктор:

```
<?php
class User
{
    public $name;
    public $age;
    // Конструктор объекта:
    public function __construct($name, $age)
    {
        $this->name = $name; // запишем данные в свойство name
        $this->age = $age; // запишем данные в свойство age
    }
}
$user = new User('john', 25); // создадим объект, сразу заполнив его
данными
echo $user->name; // выведет 'john'
```

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

Свойства только для чтения в ООП на PHP

Сейчас мы с вами сделаем так, чтобы в объекте какое-то свойство было доступно только для чтения, но не для записи (англ. read-only).

Это делается следующим образом: для такого свойства нужно сделать геттер, но не делать сеттер. В этом случае свойство можно будет прочитать с помощью геттера, но нельзя будет записать, так как сеттер отсутствует. При этом изначальное значение свойства будет задаваться в конструкторе при создании объекта.

Попробуем реализовать описанное.

Пусть у нас дан вот такой класс User:

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

Давайте сделаем так, чтобы свойство name было доступно только для чтения, а свойство age - и для чтения и для записи. Для этого свойству name сделаем только геттер, а свойству age - и геттер и сеттер:

```
<?php
class User
{
    private $name;
    private $age;
    // Геттер для имени:
    public function getName()
    {
        return $this->name;
    }
    // Геттер для возраста:
    public function getAge()
    {
        return $this->age;
    }
    // Сеттер для возраста:
    public function setAge($age)
    {
        $this->age = $age;
    }
}
?>
```

Давайте теперь добавим конструктор объекта, в котором будем задавать начальные значения наших свойств:

```
<?php
class User
{
    private $name;
    private $age;
    // Конструктор объекта:
    public function __construct($name, $age)
    {
        $this->name = $name;
        $this->age = $age;
    }
    // Геттер для имени:
    public function getName()
    {
        return $this->name;
    }
    // Геттер для возраста:
    public function getAge()
    {
        return $this->age;
    }
    // Сеттер для возраста:
    public function setAge($age)
    {
        $this->age = $age;
    }
}
?>
```

Все - наша задача решена, убедимся в этом:

```
<?php
$user = new User('john', 25); // создаем объект с начальными данными
// Имя можно только читать, но нельзя поменять:
echo $user->getName(); // выведет 'john'
// Возраст можно и читать, и менять:
echo $user->getAge(); // выведет 25
echo $user->setAge(30); // установим возраст в значение 30
echo $user->getAge(); // выведет 30
?>
```

Хранение классов в отдельных файлах в PHP

До этого урока мы писали наши классы в том же файле, где и вызывали их. В реальной жизни классы обычно хранятся в отдельных файлах, причем каждый класс в своем отдельном файле. При этом существует соглашение о том, что файл с классом следует называть так же, как и сам класс. Давайте посмотрим на практике. Сделаем файл User.php с классом User:

```
<?php
class User
{
}
?>
```

Пусть теперь у нас есть файл index.php, в котором мы хотим воспользоваться нашим классом User. Мы не можем в этом файле просто взять и создать объект класса User - это вызовет ошибку, так как PHP не сможет найди код этого класса:

```
<?php
$user = new User; // это вызовет ошибку
?>
```

Для того, чтобы класс User был доступен в файле index.php, нужно подключить к нему файл с нашим классом. Сделаем это с помощью команды require_once:

```
<?php
require_once 'User.php'; // подключаем наш класс
$user = new User;
?>
```

Хранение объектов в массивах в ООП на PHP

Пусть у нас дан вот такой класс User:

```
<?php
class User
{
    public $name;
    public $age;
    public function __construct($name, $age)
    {
        $this->name = $name;
        $this->age = $age;
    }
}
?>
```

Подключим файл с нашим классом к файлу index.php:

```
<?php
require_once 'User.php';
?>
```

Создадим три объекта нашего класса:

```
<?php
$user1 = new User('john', 21);
$user2 = new User('eric', 22);
$user3 = new User('kyle', 23);
?>
```

Давайте теперь запишем созданные нами объекты в массив $users:

```
<?php
$user1 = new User('john', 21);
$user2 = new User('eric', 22);
$user3 = new User('kyle', 23);
$users[] = $user1;
$users[] = $user2;
$users[] = $user3;
var_dump($users);
?>
```

В общем-то переменные, в которые мы сохраняем наши объекты, и не нужны. Можем сократить наш код:

```
<?php
$users[] = new User('john', 21);
$users[] = new User('eric', 22);
$users[] = new User('kyle', 23);
var_dump($users);
?>
```

Давайте теперь переделаем наш код в другом стиле - добавим элементы в массив сразу при его создании:

```
<?php
$user1 = new User('john', 21);
$user2 = new User('eric', 22);
$user3 = new User('kyle', 23);
$users = [$user1, $user2, $user3];
var_dump($users);
?>
```

Здесь также можно избавится от промежуточных переменных:

```
<?php
$users = [
    new User('john', 21),
    new User('eric', 22),
    new User('kyle', 23)
];
var_dump($users);
?>
```

Неважно каким способом мы создаем наш массив с объектами - важен сам принцип: объекты можно хранить в массивах. Затем эти объекты можно, к примеру, перебрать циклом. Давайте сделаем это:

```php
<?php
$users = [
    new User('john', 21),
    new User('eric', 22),
    new User('kyle', 23)
];
// Переберем созданный массив циклом:
foreach ($users as $user) {
    echo $user->name . ' ' . $user->age . '<br>';
}
?>
```

Начальные значения свойств в конструкторе

Пусть у нас есть какой-то класс с двумя свойствами:

```php
<?php
class Test
{
    public $prop1;
    public $prop2;
}
?>
```

Давайте сделаем так, чтобы при создании объекта класса эти свойства имели какие-либо значения. Как вы уже знаете, в момент создания объекта вызывается метод-конструктор. Зададим начальные значения свойства в этом методе:

```php
<?php
class Test
{
    public $prop1;
    public $prop2;
    public function __construct()
    {
        $this->prop1 = 'value1'; // начальное значение свойства
        $this->prop2 = 'value2'; // начальное значение свойства
    }
}
$test = new Test;
echo $test->prop1; // выведет 'value1'
echo $test->prop2; // выведет 'value2'
?>
```

Применение

Пусть у нас есть класс Student с двумя свойствами - name и course (курс студента). Сделаем так, чтобы имя студента приходило параметром при создании объекта, а курс автоматически принимал значение 1:

```php
<?php
class Student
{
    private $name;
    private $course;
    public function __construct($name)
    {
        $this->name = $name;
        $this->course = 1; // курс изначально равен 1
    }
}
?>
```

Сделаем геттеры для наших свойств:

```php
<?php
class Student
{
    private $name;
    private $course;
    public function __construct($name)
    {
        $this->name = $name;
        $this->course = 1;
    }
    // Геттер имени:
    public function getName()
    {
        return $this->name;
    }
    // Геттер курса:
    public function getCourse()
    {
        return $this->course;
    }
}
?>
```

Пусть имя созданного студента будет неизменяемым и доступным только для чтения, а вот для курса мы сделаем метод, который будет переводить нашего студента на следующий курс:

```php
<?php
class Student
{
    private $name;
    private $course;
    public function __construct($name)
    {
        $this->name = $name;
        $this->course = 1;
    }
    // Геттер имени:
    public function getName()
    {
        return $this->name;
    }
    // Геттер курса:
    public function getCourse()
    {
        return $this->course;
    }
    // Перевод студента на новый курс:
    public function transferToNextCourse()
    {
        $this->course++;
    }
}
?>
```

Проверим работу нашего класса:

```php
<?php
$student = new Student('john'); // создаем объект класса
echo $student->getCourse(); // выведет 1 - начальное значение
$student->transferToNextCourse(); // переведем студента на следующий
курс
echo $student->getCourse(); // выведет 2
?>
```

Начальные значения свойств при объявлении

Рассмотрим следующий класс:

```
<?php
class Test
{
    public $prop1;
    public $prop2;
    public function __construct()
    {
        $this->prop1 = 'value1'; // начальное значение свойства prop1
        $this->prop2 = 'value2'; // начальное значение свойства prop2
    }
}
$test = new Test;
echo $test->prop1; // выведет 'value1'
echo $test->prop2; // выведет 'value2'
?>
```

Как вы видите, в этом коде в конструкторе объекта мы задаем начальные значения свойств. На самом деле можно сократить лишний код, задав начальные значения свойств прямо при их объявлении:

```
<?php
class Test
{
    public $prop1 = 'value1'; // начальное значение свойства prop1
    public $prop2 = 'value2'; // начальное значение свойства prop2
}
$test = new Test;
echo $test->prop1; // выведет 'value1'
echo $test->prop2; // выведет 'value2'
?>
```

Замечания

Конечно же, не обязательно задавать начальные значения всем свойствам:

```
<?php
class Test
{
    public $prop1 = 'value1'; // задаем начальное значение
    public $prop2; // не задаем
}
?>
```

При задании начальных значений свойств можно выполнять некоторые операции (самые примитивные):

```
<?php
class Test
{
    public $prop = 1 + 2; // найдем сумму чисел
}
$test = new Test;
echo $test->prop; // выведет 3
?>
```

Применение

Пусть у нас есть вот такой класс Student, в конструкторе которого задается начальное значение свойства course:

```
<?php
class Student
{
    private $name;
    private $course;
    public function __construct($name)
    {
        $this->name = $name;
        $this->course = 1; // начальное значение курса
    }
    public function transferToNextCourse()
    {
        $this->course++;
    }
}
?>
```

Давайте вынесем начальное значение курса в объявление свойства:

```
<?php
class Student
{
    private $name;
    private $course = 1; // начальное значение курса
    public function __construct($name)
    {
        $this->name = $name;
    }
    public function transferToNextCourse()
    {
        $this->course++;
    }
}
?>
```

Применение

Пусть у нас есть вот такой класс Arr, у которого есть метод add для добавления чисел и метод getSum для получения суммы всех добавленных чисел:

```
<?php
class Arr
{
    // Массив для хранения чисел:
    private $numbers;
    // Добавляет число в набор:
    public function add($num)
    {
        $this->numbers[] = $num;
    }
    // Находит сумму чисел набора:
    public function getSum()
    {
        return array_sum($this->numbers);
    }
}
?>
```

Давайте воспользуемся нашим классом Arr - добавим несколько чисел и найдем их сумму:

```
<?php
$arr = new Arr;
$arr->add(1);
$arr->add(2);
$arr->add(3);
echo $arr->getSum(); // выведет 6
?>
```

Все вроде работает, но что будет, если сразу после создания вызвать метод getSum? Вот таким образом:

```
<?php
$arr = new Arr;
echo $arr->getSum();
?>
```

Такой код вызовет ошибку, потому что функция array_sum пытается найти сумму массива из свойства numbers. Но это свойство еще не определено и имеет значение null. Это и приводит к ошибке. Давайте исправим проблему, объявив наше свойство пустым массивом:

```
<?php
class Arr
{
    private $numbers = []; // задаем начальное значение свойства как
[]
    public function add($num)
    {
        $this->numbers[] = $num;
    }
    public function getSum()
    {
        return array_sum($this->numbers);
    }
}
?>
```

Проверим:

```
<?php
$arr = new Arr;
echo $arr->getSum(); // выведет 0
?>
```

Переменные названия свойств объектов в PHP

Пусть у нас есть вот такой класс User:

```
<?php
class User
{
    public $name;
    public $age;
    public function __construct($name, $age)
    {
        $this->name = $name;
        $this->age = $age;
    }
}
$user = new User('john', 21);
echo $user->name; // выведет 'john'
?>
```

На примере этого класса мы сейчас разберем то, что названия свойств можно хранить в переменной.

К примеру, пусть у нас есть переменная $prop, в которой лежит строка 'name'. Тогда обращение $user->$prop по сути эквивалентно обращению $user->name. Такое хитрое обращения к свойствам класса используется редко, но иногда бывает полезно.

Посмотрим на примере:

```
<?php
$user = new User('john', 21);
$prop = 'name';
echo $user->$prop; // выведет 'john'
?>
```

Массив свойств

Пусть теперь дан вот такой класс User:

```
<?php
class User
{
    public $surname; // фамилия
    public $name; // имя
    public $patronymic; // отчество
    public function __construct($surname, $name, $patronymic)
    {
        $this->surname = $surname;
        $this->name = $name;
        $this->patronymic = $patronymic;
    }
}
?>
```

И пусть дан массив свойств

```
<?php
$props = ['surname', 'name', 'patronymic'];
?>
```

Попробуем теперь вывести значение свойства, которое хранится в нулевом элементе массива, то есть в $props[0]. Просто так, однако, это не будет работать:

```
<?php
$user = new User('Иванов', 'Иван', 'Иванович');
$props = ['surname', 'name', 'patronymic'];
echo $user->$props[0]; // так работать не будет
?>
```

Для того, чтобы такое сложное имя свойства заработало, его нужно взять в фигурные скобки, вот так:

```
<?php
$user = new User('Иванов', 'Иван', 'Иванович');
$props = ['surname', 'name', 'patronymic'];
echo $user->{$props[0]}; // выведет 'Иванов'
?>
```

Ассоциативный массив

Массив, кстати, может быть и ассоциативным:

```
<?php
$user = new User('Иванов', 'Иван', 'Иванович');
$props = ['prop1' => 'surname', 'prop2' => 'name', 'prop3' =>
'patronymic'];
echo $user->{$props['prop1']}; // выведет 'Иванов'
?>
```

Имя свойства из функции

Имя свойства также может быть из функции:

```
<?php
function getProp()
{
    return 'surname';
}
$user = new User('Иванов', 'Иван', 'Иванович');
echo $user->{getProp()}; // выведет 'Иванов'
?>
```

Имя свойства из свойства другого объекта

Имя свойства может быть даже свойством другого объекта.

Проиллюстрируем кодом. Пусть для примера дан объект Prop, который в свойстве value будет содержать название свойства объекта User:

```
<?php
class Prop
{
    public $value;
    public function __construct($value)
    {
        $this->value = $value;
    }
}
?>
```

Давайте выведем фамилию юзера с помощью объекта Prop:

```
<?php
$user = new User('Иванов', 'Иван', 'Иванович');
$prop = new Prop('surname'); // будем выводить значение свойства
surname
echo $user->{$prop->value}; // выведет 'Иванов'
?>
```

Имя свойства из метода другого объекта

Имя свойства также может браться из метода другого объекта:

```
<?php
class Prop
{
    private $value;
    public function __construct($value)
    {
        $this->value = $value;
    }
    public function getValue()
    {
        return $this->value;
    }
}
?>
```

Давайте выведем фамилию юзера:

```
<?php
$user = new User('Иванов', 'Иван', 'Иванович');
$prop = new Prop('surname'); // будем выводить значение свойства
surname
echo $user->{$prop->getValue()}; // выведет 'Иванов'
?>
```

Переменные названия методов

По аналогии с названиями свойств в переменной также можно хранить и имена методов. Давайте посмотрим на примере. Пусть у нас дан вот такой класс User с геттерами свойств:

```
<?php
class User
{
    private $name;
    private $age;
    public function __construct($name, $age)
    {
        $this->name = $name;
        $this->age = $age;
    }
    public function getName()
    {
        return $this->name;
    }
    public function getAge()
    {
        return $this->age;
    }
}
?>
```

Пусть в переменной $method хранится имя метода. Давайте вызовем метод с таким именем:

```
<?php
$user = new User('john', 21);
$method = 'getName';
echo $user->$method(); // выведет 'john'
?>
```

Если имя метода получается из массива, то такое обращение к методу следует брать в фигурные скобки вот таким образом (круглые скобки будут снаружи фигурных):

```
<?php
$user = new User('john', 21);
$methods = ['getName', 'getAge'];
echo $user->{$methods[0]}(); // выведет 'john'
?>
```

Все остальные нюансы точно такие же, как и при работе со свойствами из переменной.

Вызов метода сразу после создания объекта

Пусть у нас дан класс Arr, который хранит в себе массив чисел и может вычислять сумму этих чисел с помощью метода getSum. Сами числа приходят в виде массива в конструктор объекта, а также могут добавляться по одному с помощью метода add:

```
<?php
class Arr
{
    private $numbers = []; // массив чисел
    public function __construct($numbers)
    {
        $this->numbers = $numbers; // записываем массив $numbers в
свойство
    }
    // Добавляем еще одно число в наш массив:
    public function add($number)
    {
        $this->numbers[] = $number;
    }
    // Находим сумму чисел:
    public function getSum()
    {
        return array_sum($this->numbers);
    }
}
?>
```

Вот пример использования класса Arr:

```
<?php
$arr = new Arr([1, 2, 3]); // создаем объект, записываем в него массив
[1, 2, 3]
$arr->add(4); // добавляем в конец массива число 4
$arr->add(5); // добавляем в конец массива число 5
// Находим сумму элементов массива:
echo $arr->getSum(); // выведет 15
?>
```

Может такое быть, что все нужные числа мы передадим в момент создания объекта, а затем сразу захотим найти их сумму:

```
<?php
$arr = new Arr([1, 2, 3]);
echo $arr->getSum(); // выведет 6
?>
```

Если мы больше не планируем делать никаких манипуляций с объектом, то код выше можно переписать короче: можно создать объект и сразу вызвать его метод getSum:

```
<?php
echo (new Arr([1, 2, 3]))->getSum(); // выведет 6
?>
```

Вот еще пример - найдем сумму двух массивов:

```
<?php
echo (new Arr([1, 2, 3]))->getSum() + (new Arr([4, 5, 6]))->getSum();
?>
```

Цепочки методов

Пусть у нас дан класс Arr, который хранит в себе массив чисел и может вычислять сумму этих чисел с помощью метода getSum. Числа могут добавляться по одному с помощью метода add, либо группой с помощью метода push:

```php
<?php
class Arr
{
    private $numbers = [];
    public function add($number)
    {
        $this->numbers[] = $number;
    }
    public function push($numbers)
    {
        $this->numbers = array_merge($this->numbers, $numbers);
    }
    public function getSum()
    {
        return array_sum($this->numbers);
    }
}
?>
```

Пример использования класса Arr:

```php
<?php
$arr = new Arr; // создаем объект
$arr->add(1); // добавляем в массив число 1
$arr->add(2); // добавляем в массив число 2
$arr->push([3, 4]); // добавляем группу чисел
echo $arr->getSum(); // находим сумму элементов массива
?>
```

Пусть теперь мы хотим сделать так, чтобы методы вызывались не отдельно, а цепочкой, вот так:

```php
<?php
echo $arr->add(1)->add(2)->push([3, 4])->getSum(); // это наша цель
?>
```

Для того, чтобы можно было написать такую цепочку, нужно, чтобы все методы, которые участвуют в цепочке возвращали $this (кроме последнего). Поправим наш класс Arr:

```php
<?php
class Arr
{
    private $numbers = []; // массив чисел
    public function add($number)
    {
        $this->numbers[] = $number;
        return $this; // вернем ссылку сами на себя
    }
    public function push($numbers)
    {
        $this->numbers = array_merge($this->numbers, $numbers);
        return $this; // вернем ссылку сами на себя
    }
    public function getSum()
    {
        return array_sum($this->numbers);
    }
}
?>
```

Проверим, что все работает:

```php
<?php
$arr = new Arr;
echo $arr->add(1)->add(2)->push([3, 4])->getSum(); // выведет 10
?>
```

Можно упростить еще больше:

```php
<?php
echo (new Arr)->add(1)->add(2)->push([3, 4])->getSum(); // выведет 10
?>
```

Класс как набор методов в ООП на PHP

Часто классы используются просто как набор некоторых методов, сгруппированных вместе. В этом случае нам не нужно создавать много объектов этого класса, а достаточно всего одного.

Для примера давайте сделаем класс ArraySumHelper, который предоставит нам набор методов для работы с массивами. Каждый метод нашего класса будет принимать массив, что-то с ним делать и возвращать результат. Пусть у нас будет следующий набор методов:

```
<?php
class ArraySumHelper
{
    // Сумма элементов массива:
    public function getSum1($arr)
    {
    }
    // Сумма квадратов элементов массива:
    public function getSum2($arr)
    {
    }
    // Сумма кубов элементов массива:
    public function getSum3($arr)
    {
    }
    // Сумма четвертых степеней элементов массива:
    public function getSum4($arr)
    {
    }
}
?>
Давайте посмотрим, как мы будем пользоваться нашим классом:

<?php
$arraySumHelper = new ArraySumHelper;
$arr = [1, 2, 3];
echo $arraySumHelper->getSum1($arr);
echo $arraySumHelper->getSum2($arr);
echo $arraySumHelper->getSum3($arr);
echo $arraySumHelper->getSum4($arr);
?>
```

Вот еще пример - найдем сумму квадратов элементов массива и сумму кубов и сложим результат вместе:

```
<?php
$arraySumHelper = new ArraySumHelper;
$arr = [1, 2, 3];
echo $arraySumHelper->getSum2($arr) + $arraySumHelper->getSum3($arr);
?>
```

Фактически мы получаем набор функций, просто сгруппированных в одном классе. Однако, в отличие от обычного набора функций, мы можем пользоваться преимуществами ООП - например, делать вспомогательные методы приватными, чтобы они не были доступны извне класса.

Давайте приступим к написанию кода нашего класса.

Если обдумать реализацию методов, то становится очевидно, что они будут фактически одинаковыми, отличие будет только в степени, в которую будут возводится элементы нашего массива. Вот код, иллюстрирующий это:

```
<?php
class ArraySumHelper
{
    public function getSum1($arr)
    {
        $sum = 0;
        foreach ($arr as $elem) {
            $sum += $elem; // первая степень элемента - это сам
элемент
        }
        return $sum;
    }
    public function getSum2($arr)
    {
        $sum = 0;
        foreach ($arr as $elem) {
            $sum += pow($elem, 2); // возведем во вторую степень
        }
        return $sum;
    }
    public function getSum3($arr)
    {
        $sum = 0;
        foreach ($arr as $elem) {
            $sum += pow($elem, 3); // возведем в третью степень
        }
        return $sum;
    }
    public function getSum4($arr)
    {
        $sum = 0;
        foreach ($arr as $elem) {
            $sum += pow($elem, 4); // возведем в четвертую степень
        }
        return $sum;
    }
}
?>
```

Вместо того, чтобы реализовывать каждый метод заново, сделаем вспомогательный приватный метод getSum, который параметрами будет принимать массив и степень и возвращать сумму степеней элементов массива:

```
<?php
private function getSum($arr, $power) {
    $sum = 0;
    foreach ($arr as $elem) {
        $sum += pow($elem, $power);
    }
    return $sum;
}
?>
```

Давайте поменяем методы нашего класса с использованием нового метода getSum:

```
<?php
class ArraySumHelper
{
    public function getSum1($arr)
    {
        return $this->getSum($arr, 1);
    }
    public function getSum2($arr)
    {
        return $this->getSum($arr, 2);
    }
    public function getSum3($arr)
    {
        return $this->getSum($arr, 3);
    }
    public function getSum4($arr)
    {
        return $this->getSum($arr, 4);
    }
    private function getSum($arr, $power) {
        $sum = 0;
        foreach ($arr as $elem) {
            $sum += pow($elem, $power);
        }
        return $sum;
    }
}
?>
```

Наш класс ArraySumHelper больше учебный, чем реальный, но тут вам важно понять принцип - то, что часто некоторый класс может использоваться просто как набор методов и при этом создается только один объект класса. В дальнейшем мы будем разбирать более жизненные (но и более сложные) примеры.

Наследование классов

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

Вот этот класс:

```
<?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;
    }
}
?>
```

А теперь представим себе ситуацию, когда нам понадобился еще и класс Employee. Работник очень похож на юзера, имеет те же свойства и методы, но еще и добавляет свои - свойство salary, а также геттер и сеттер для этого свойства. Вот этот класс:

```php
<?php
class Employee
{
    private $name;
    private $age;
    private $salary; // зарплата
    // Геттер зарплаты
    public function getSalary()
    {
        return $this->salary;
    }
    // Сеттер зарплаты
    public function setSalary($salary)
    {
        $this->salary = $salary;
    }
    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;
    }
}
?>
```

Как мы видим, код классов User и Employee практически полностью совпадает. Было бы намного лучше сделать так, чтобы общая часть была записана только в одном месте.

Для решения проблемы существует такая вещь, как наследование. С помощью наследования мы можем заставить наш класс Employee позаимствовать (унаследовать) методы и свойства класса User и просто дополнить их своими методами и свойствами.

Класс, от которого наследуют называется родителем (англ. parent), а класс, который наследует - потомком. Класс-потомок наследует только публичные методы и свойства, но не приватные.

Наследование реализуется с помощью ключевого слова extends (переводится как расширяет). Перепишем наш класс Employee так, чтобы он наследовал от User:

```
<?php
class Employee extends User
{
    private $salary;
    public function getSalary()
    {
        return $this->salary;
    }
    public function setSalary($salary)
    {
        $this->salary = $salary;
    }
}
?>
```

Проверим работу нового класса Employee:

```
<?php
$employee = new Employee;
$employee->setSalary(1000); // метод класса Employee
$employee->setName('john'); // метод унаследован от родителя
$employee->setAge(25); // метод унаследован от родителя
echo $employee->getSalary(); // метод класса Employee
echo $employee->getName(); // метод унаследован от родителя
echo $employee->getAge(); // метод унаследован от родителя
?>
```

Класс-потомок не унаследовал от своего родителя приватные свойства name и age - попытка обратится к ним вызовет ошибку. При этом, однако, в классе-потомке доступны геттеры и сеттеры этих свойств, так как эти геттеры и сеттеры являются публичными.

Несколько потомков

Преимущества наследования в том, что каждый класс может иметь несколько потомков.

Посмотрим на примере. Пусть кроме работника нам нужен еще и класс Student - давайте также 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;
    }
}
?>
```

Проверим работу нашего класса:

```
<?php
$student = new Student;
$student->setCourse(3); // метод класса Student
$student->setName('john'); // метод унаследован от родителя
$student->setAge(25); // метод унаследован от родителя
echo $student->getCourse(); // метод класса Student
echo $student->getName(); // метод унаследован от родителя
echo $student->getAge(); // метод унаследован от родителя
?>
```

Наследование от наследников

Пусть у нас есть класс-родитель и класс-потомок. От этого потомка также могут наследовать другие классы, от его потомков другие и так далее. Для примера пусть от класса User наследует Student, а от него в свою очередь наследует класс StudentBSU:

```
<?php
class StudentBSU extends Student
{
    // добавляем свои свойства и методы
}
?>
```

Модификатор доступа protected в ООП на PHP

Как вы уже знаете, приватные свойства и методы не наследуются. Если вы хотите, чтобы метод или свойство появились у потомка, вы должны объявить их как public. Проблема, однако, в том, что публичные методы будут также доступны и извне класса, а мы бы этого не хотели.

Другими словами, мы хотели бы, чтобы некоторые методы или свойства родителя наследовались потомками, но при этом для всего остального мира вели себя как приватные.

Для решения этой проблемы существует специальный модификатор protected, который и реализует описанное.

Изучим его работу на реальном примере.

Пусть у нас дан вот такой класс User с приватными свойствами name и age:

```
<?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;
    }
}
?>
```

Пусть от класса User наследует класс Student:

```
<?php
class Student extends User
{
    private $course;
    public function getCourse()
    {
        return $this->course;
    }
    public function setCourse($course)
    {
        $this->course = $course;
    }
}
?>
```

Пока все отлично и все работает.

Допустим теперь мы решили в классе Student сделать метод addOneYear, который будет добавлять 1 год к свойству age. Давайте реализуем указанный метод:

```
<?php
class Student extends User
{
    private $course;
    // Реализуем этот метод:
    public function addOneYear()
    {
        $this->age++;
    }
    public function getCourse()
    {
        return $this->course;
    }
    public function setCourse($course)
    {
        $this->course = $course;
    }
}
?>
```

Проблема в том, что если оставить свойство age приватным, то мы не сможем обратится к нему в классе-потомке - это выдаст ошибку при попытке вызова метода addOneYear:

```
<?php
$student = new Student();
$student->setAge(25);
$student->addOneYear(); // выдаст ошибку
?>
```

Для исправления ошибки поправим класс User - объявим свойство age как protected, а не как private:

```
<?php
class User
{
    private $name;
    protected $age; // объявим свойство как protected
...
}
?>
```

Теперь метод addOneYear потомка сможет менять свойство age, но оно по-прежнему не будет доступно извне наших классов. Проверим работу класса Student:

```
<?php
$student = new Student();
$student->setName('john'); // установим имя
$student->setCourse(3);    // установим курс
$student->setAge(25);      // установим возраст в 25
$student->addOneYear();    // увеличим возраст на единицу
echo $student->getAge();   // выведет 26
?>
```

Попытка обратится к свойству age снаружи класса выдаст ошибку, как нам и нужно:

```
<?php
$student = new Student();
$student->age = 30; // выдаст ошибку
?>
```

Перезапись методов родителя в классе потомке

Пусть дан класс User с приватными свойствами name и age, а также геттерами и сеттерами этих свойств. При этом в сеттере возраста выполняется проверка возраста на то, что он равен или больше 18 лет:

```
<?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)
    {
        if ($age >= 18) {
            $this->age = $age;
        }
    }
}
?>
```

От класса User наследует класс Student с приватным свойством course, его геттером и сеттером:

```
<?php
class Student extends User
{
    private $course;
    public function getCourse()
    {
        return $this->course;
    }
    public function setCourse($course)
    {
        $this->course = $course;
    }
}
?>
```

Предположим теперь, что метод setAge, который Student наследует от User нам чем-то не подходит, например, нам нужна также проверка того, что возраст студента до 25 лет.

То есть проверка на то, что возраст более 18 лет нас устраивает, но хотелось бы иметь добавочную проверку на то, что он меньше 25.

Для решения проблемы мы используем то, что в PHP в классе-потомке разрешено сделать метод с таким же именем, как и у родителя, таким образом переопределив этот метод родителя на свой.

Как раз то, что нам нужно.

Итак, пишем свой setAge в классе Student. Наш метод будет проверять то, что возраст студента от 18 до 25 лет:

```
<?php
class Student extends User
{
    private $course;
    // Перезаписываем метод родителя:
    public function setAge($age)
    {
        if ($age >= 18 and $age <= 25) {
            $this->age = $age;
        }
    }
    public function getCourse()
    {
        return $this->course;
    }
    public function setCourse($course)
    {
        $this->course = $course;
    }
}
?>
```

Так как наш метод setAge использует свойство age от родителя, то в родителе это свойство надо объявить как защищенное:

```
<?php
class User
{
    private $name;
    protected $age; // изменим модификатор доступа на protected
    public function getName()
    {
        return $this->name;
    }
    public function setName($name)
    {
        $this->name = $name;
    }
    public function getAge()
    {
        return $this->age;
    }
    public function setAge($age)
    {
        if ($age >= 18) {
            $this->age = $age;
        }
    }
}
?>
```

Теперь проверим работу переопределенного метода setAge:

```
<?php
$student = new Student;
$student->setAge(24);    // укажем корректный возраст
echo $student->getAge(); // выведет 24 - возраст поменялся
$student->setAge(30);    // укажем некорректный возраст
echo $student->getAge(); // выведет 24 - возраст не поменялся
?>
```

Работа с parent

Сейчас в нашем новом методе setAge мы выполняем проверку того, что возраст от 18 до 25 лет. Однако, проверку, того, что возраст от 18 лет выполняет и метод setAge родителя. Это значит, что если мы захотим изменить нижнюю границу возраста - нам придется сделать это в двух местах: в коде класса родителя и в коде класса потомка.

Было бы удобно, если мы метод setAge потомка мог использовать метод setAge от родителя, ведь в методе родителя расположена часть кода, которая нам подходит и мы не хотим ее дублировать в методе потомка.

Такое можно сделать с помощью ключевого слова parent, указывающего на родителя. С его помощью к переопределенному методу родителя можно

обратиться так: `parent::setAge(),` то есть ключевое слово parent, затем два двоеточия и сам метод.

Давайте доработаем наш класс Student так, чтобы использовался метод setAge родителя:

```
<?php
class Student extends User
{
    private $course;
    public function setAge($age)
    {
        // Если возраст меньше или равен 25:
        if ($age <= 25) {
            // Вызываем метод родителя:
            parent::setAge($age); // в родителе выполняется проверка
age >= 18
        }
    }
    public function getCourse()
    {
        return $this->course;
    }
    public function setCourse($course)
    {
        $this->course = $course;
    }
}
?>
```

Мы добились того, что хотели. Более того, теперь метод setAge потомка не использует свойство age напрямую. Это значит, что в классе-родителе можно поменять его модификатор доступа с protected обратно на private.

Перезапись конструктора родителя в потомке

Пусть у нас есть вот такой класс User, у которого свойства name и age задаются в конструкторе и в дальнейшем доступны только для чтения (то есть приватные и имеют только геттеры, но не сеттеры):

```php
<?php
class User
{
    private $name;
    private $age;
    public function __construct($name, $age)
    {
        $this->name = $name;
        $this->age = $age;
    }
    public function getName()
    {
        return $this->name;
    }
    public function getAge()
    {
        return $this->age;
    }
}
?>
```

От этого класса наследует класс Student:

```php
<?php
class Student extends User
{
    private $course;
    public function getCourse()
    {
        return $this->course;
    }
}
?>
```

Класс-потомок не имеет своего конструктора - это значит что при создании объекта класса сработает конструктор родителя:

```php
<?php
$student = new Student('john', 19); // сработает конструктор родителя
echo $student->getName(); // выведет 'john'
echo $student->getAge();  // выведет 19
?>
```

Все замечательно, но есть проблема: мы бы хотели при создании объекта класса Student третьим параметром передавать еще и курс, вот так:

```php
<?php
$student = new Student('john', 19, 2); // это пока не работает
?>
```

Самое простое, что можно сделать, это переопределить конструктор родителя своим конструктором, забрав из родителя его код:

```php
<?php
class Student extends User
{
    private $course;
    // Конструктор объекта:
    public function __construct($name, $age, $course)
    {
        // Дублируем код конструктора родителя:
        $this->name = $name;
        $this->age = $age;
        // Наш код:
        $this->course = $course;
    }
    public function getCourse()
    {
        return $this->course;
    }
}
?>
```

При этом мы в классе потомке обращаемся к приватным свойствам родителя name и age, что, конечно же, не будет работать так, как нам нужно. Переделаем их на protected:

```php
<?php
class User
{
    protected $name; // объявим свойство защищенным
    protected $age;  // объявим свойство защищенным
    // Конструктор объекта:
    public function __construct($name, $age)
    {
        $this->name = $name;
        $this->age = $age;
    }
    public function getName()
    {
        return $this->name;
    }
    public function getAge()
    {
        return $this->age;
    }
}
?>
```

Теперь при создании студента третьим параметром мы можем передать и курс:

```php
<?php
$student = new Student('john', 19, 2); // теперь это работает
echo $student->getName();   // выведет 'john'
echo $student->getAge();    // выведет 19
echo $student->getCourse(); // выведет 2
?>
```

Используем конструктор родителя

Понятно, что дублирование кода родителя в классе потомке - это не очень хорошо. Давайте вместо дублирования кода в конструкторе потомка вызовем конструктор родителя.

Для полной ясности распишем все по шагам.

Вот так выглядит конструктор класса User, он принимает два параметра $name и $age и записывает их в соответствующие свойства:

```
<?php
// Конструктор объекта класса User:
public function __construct($name, $age)
{
    $this->name = $name;
    $this->age  = $age;
}
?>
```

Вот конструктор класса Student, который мы хотим переписать:

```
<?php
// Конструктор объекта класса Student:
public function __construct($name, $age, $course)
{
    // Этот код хотим заменить на вызов конструктора родителя:
    $this->name = $name;
    $this->age  = $age;
    // Наш код:
    $this->course = $course;
}
?>
```

Конструктор родителя можно вызвать внутри потомка с помощью parent. При этом конструктор родителя первым параметром ожидает имя, а вторым - возраст, и мы должны ему их передать, вот так:

```
<?php
// Конструктор объекта класса Student:
public function __construct($name, $age, $course)
{
    // Вызовем конструктор родителя, передав ему два параметра:
    parent::__construct($name, $age);
    // Запишем свойство course:
    $this->course = $course;
}
?>
```

Напишем полный код класса Student:

```
<?php
class Student extends User
{
    private $course;
    // Конструктор объекта:
    public function __construct($name, $age, $course)
    {
        parent::__construct($name, $age); // вызываем конструктор
родителя
        $this->course = $course;
    }
    public function getCourse()
    {
        return $this->course;
    }
}
?>
```

Проверим, что все работает:

```
<?php
$student = new Student('john', 19, 2);
echo $student->getName();   // выведет 'john'
echo $student->getAge();    // выведет 19
echo $student->getCourse(); // выведет 2
?>
```

Так как класс Student теперь не обращается напрямую к свойствам name и age родителя, можно их опять сделать приватными:

```
<?php
class User
{
    private $name; // объявим свойство приватным
    private $age;  // объявим свойство приватным
    public function __construct($name, $age)
    {
        $this->name = $name;
        $this->age  = $age;
    }
    public function getName()
    {
        return $this->name;
    }
    public function getAge()
    {
        return $this->age;
    }
}
?>
```

Передача объектов по ссылке

Пусть у нас дан вот такой класс User:

```
<?php
class User
{
    public $name;
    public $age;
    public function __construct($name, $age)
    {
        $this->name = $name;
        $this->age  = $age;
    }
}
?>
```

Пусть мы создаем объект этого класса:

```
<?php $user = new User('john', 30); ?>
```

Представьте теперь следующую ситуацию: вы хотите значение переменной $user присвоить какой-нибудь другой переменной, например $test.

Если речь шла не об объектах, а о примитивах, то есть о строках, числах, массивах и тп, то в переменную $test попала бы копия значения переменной $user.

Это значит, что изменения любой из переменной в дальнейшем не меняли бы значение другой переменной. Посмотрим на примере:

```
<?php
$user = 1;
$test = $user; // в переменной $test теперь 1
test = 2;     // в переменной $test теперь 2, а в $user по-прежнему 1
?>
```

С объектами все по-другому - при записи в другую переменную объекты не копируются, а передаются по ссылке: это значит, что обе эти переменные значением имеют один и тот же объект. Это имеет важное следствие: если поменять какие-то свойства объекта с помощью одной переменной, во второй переменной появятся эти же изменения:

```
<?php
$user = new User('john', 30);
$test = $user; // и $test, и $user ссылаются на один и тот же объект
$test->name = 'eric'; // поменяли переменную $test - но $user также
поменялась!
// Проверим - выведем свойство name из переменной $user:
echo $user->name; // выведет 'eric'!
?>
```

Учтите, что речь идет именно об объектах. Если вы в какую-то переменную запишите данные из свойства объекта - они скопируются, а не передадутся по ссылке:

```
<?php
$user = new User('john', 30);
$name = $user->name; // запишем в переменную $name текст 'john'
$name = 'eric'; // поменяли переменную $name, но $user->name не
поменялось
// Проверим - выведем свойство name из переменной $user:
echo $user->name; // выведет 'john'
?>
```

Если перезаписать переменную с объектом на примитив, то объект не исчезнет из другой переменной:

```
<?php
$user = new User('john', 30);
$test = $user; // и $test, и $user ссылаются на один и тот же объект
$user = 123;   // теперь $test ссылается, а $user - нет
?>
```

Объект существует в памяти компьютера до тех пор, пока на него ссылается хоть кто-нибудь. В примере выше, если что-нибудь записать и в переменную $user - на наш объект больше не будет ссылаться ни одна переменная и этот объект удалится из памяти.

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.

Почему мы не хотим наследовать?

Во-первых, используемый класс может являться вспомогательным и по логике нашего кода может не подходить на роль родителя.

Во-вторых, мы можем захотеть использовать несколько классов внутри другого класса, а с наследованием это не получится, ведь в PHP у класса может быть только один родитель.

Давайте посмотрим на практическом примере.

Пусть у нас дан следующий класс Arr, в объект которого мы можем добавлять числа с помощью метода add:

```
<?php
class Arr
{
    private $nums = []; // массив чисел
    // Добавляем число в массив:
    public function add($num)
    {
        $this->nums[] = $num;
    }
}
?>
```

Давайте теперь добавим в наш класс метод, который будет находить сумму квадратов элементов и прибавлять к ней сумму кубов элементов.

Пусть у нас уже существует класс SumHelper, имеющий методы для нахождения сумм элементов массивов:

```
<?php
class SumHelper
{
    // Сумма квадратов:
    public function getSum2($arr)
    {
        return $this->getSum($arr, 2);
    }
    // Сумма кубов:
    public function getSum3($arr)
    {
        return $this->getSum($arr, 3);
    }
    // Вспомогательная функция для нахождения суммы:
    private function getSum($arr, $power) {
        $sum = 0;
        foreach ($arr as $elem) {
            $sum += pow($elem, $power);
        }
        return $sum;
    }
}
?>
```

Логично будет не реализовывать нужные нам методы еще раз в классе Arr, а воспользоваться методами класса SumHelper внутри класса Arr.

Для этого в классе Arr внутри конструктора создадим объект класса SumHelper и запишем его в свойство sumHelper:

```
<?php
class Arr
{
    private $nums = []; // массив чисел
    private $sumHelper; // сюда запишется объект класса SumHelper
    // Конструктор класса:
    public function __construct()
    {
        // Запишем объект вспомогательного класса в свойство:
        $this->sumHelper = new SumHelper;
    }
    // Добавляем число в массив:
    public function add($num)
    {
        $this->nums[] = $num;
    }
}
<?
```

Теперь внутри Arr доступно свойство $this->sumHelper, в котором хранится объект класса SumHelper с его публичными методами и свойствами (если бы публичные свойства были, сейчас их там нет, только публичные методы).

Создадим теперь в классе Arr метод getSum23, который будет находить сумму квадратов элементов и прибавлять к ней сумму кубов элементов, используя методы класса SumHelper:

```
<?php
class Arr
{
    private $nums = [];
    private $sumHelper;
    public function __construct()
    {
        $this->sumHelper = new SumHelper;
    }
    // Находим сумму квадратов и кубов элементов массива:
    public function getSum23()
    {
        // Для краткости запишем $this->nums в переменную:
        $nums = $this->nums;
        // Найдем описанную сумму:
        return $this->sumHelper->getSum2($nums) + $this->sumHelper-
>getSum3($nums);
    }
    public function add($number)
    {
        $this->nums[] = $number;
    }
}
?>
```

Теперь воспользуемся созданным классом Arr:

```
<?php
$arr = new Arr(); // создаем объект
$arr->add(1); // добавляем в массив число 1
$arr->add(2); // добавляем в массив число 2
$arr->add(3); // добавляем в массив число 3
// Находим сумму квадратов и кубов:
echo $arr->getSum23();
?>
```

Передача объектов параметрами

Пусть у нас дан вот такой класс Employee:

```
<?php
class Employee
{
    private $name;
    private $salary;
    public function __construct($name, $salary)
    {
        $this->name = $name;
        $this->salary = $salary;
    }
    public function getName()
    {
        return $this->name;
    }
    public function getSalary()
    {
        return $this->salary;
    }
}
?>
```

Давайте сделаем еще и класс EmployeesCollection, который будет хранить массив работников, то есть массив объектов класса Employee.

Пусть работники будут храниться в свойстве employees, а для добавления работников будет существовать метод add. Этот метод параметром будет принимать объект класса Employee и записывать его в конец массива $this->employees:

```
<?php
class EmployeesCollection
{
    private $employees = []; // массив работников, по умолчанию пустой
    // Добавляем нового работника:
    public function add($employee)
    {
        $this->employees[] = $employee; // $employee - объект класса
Employee
    }
}
?>
```

Давайте также добавим в наш класс метод getTotalSalary, который будет находить суммарную зарплату всех хранящихся работников:

```
<?php
class EmployeesCollection
{
    private $employees = [];
    public function add($employee)
    {
        $this->employees[] = $employee;
    }
    // Находим суммарную зарплату:
    public function getTotalSalary()
    {
        $sum = 0;
        // Перебираем работников циклом:
        foreach ($this->employees as $employee) {
            $sum += $employee->getSalary(); // получаем з/п работника
через метод getSalary()
        }
        return $sum;
    }
}
?>
```

Давайте проверим работу класса EmployeesCollection:

```
<?php
$employeesCollection = new EmployeesCollection;
$employeesCollection->add(new Employee('john', 100));
$employeesCollection->add(new Employee('eric', 200));
$employeesCollection->add(new Employee('kyle', 300));
echo $employeesCollection->getTotalSalary(); // выведет 600
?>
```

Давайте сделаем наш класс EmployeesCollection более жизненным и добавим метод получения всех работников и метод подсчета:

```php
<?php
class EmployeesCollection
{
    private $employees = [];
    // Получаем всех работников в виде массива:
    public function get()
    {
        return $this->employees;
    }
    // Подсчитываем количество хранимых работников:
    public function count()
    {
        return count($this->employees);
    }
    public function add($employee)
    {
        $this->employees[] = $employee;
    }
    public function getTotalSalary()
    {
        $sum = 0;
        foreach ($this->employees as $employee) {
            $sum += $employee->getSalary();
        }
        return $sum;
    }
}
?>
```

Сравнение объектов в ООП на PHP

Сейчас мы с вами научимся сравнивать объекты с помощью операторов == и ===.

Вы должны уже знать, что для примитивов (то есть не объектов) оператор == сравнивает данные по значению без учета типа, а оператор === - учитывая тип:

```php
<?php
var_dump(3 == 3);    // выведет true
var_dump(3 == '3');  // выведет true
var_dump(3 === 3);   // выведет true
var_dump(3 === '3'); // выведет false
?>
```

Давайте теперь посмотрим, как работает сравнение объектов.

При использовании оператора == для сравнения двух объектов выполняется сравнение свойств объектов: два объекта равны, если они имеют одинаковые свойства и их значения (значения свойств сравниваются через ==) и являются экземплярами одного и того же класса.

При сравнении через ===, переменные, содержащие объекты, считаются равными только тогда, когда они ссылаются на один и тот же экземпляр одного и того же класса.

Давайте посмотрим на примере. Пусть у нас дан вот такой класс User:

```
<?php
class User
{
    private $name;
    private $age;
    public function __construct($name, $age)
    {
        $this->name = $name;
        $this->age  = $age;
    }
    public function getName()
    {
        return $this->name;
    }
    public function getAge()
    {
        return $this->age;
    }
}
?>
```

Создадим два объекта нашего класса с одинаковыми значениями свойств и сравним созданные объекты:

```
<?php
$user1 = new User('john', 30);
$user2 = new User('john', 30);
var_dump($user1 == $user2); // выведет true
?>
```

Пусть теперь значения свойств одинаковые, но у них разный тип:

```
<?php
$user1 = new User('john', 30); // возраст - число
$user2 = new User('john', '30'); // возраст - строка
var_dump($user1 == $user2); // выведет true
?>
```

Пусть значения свойств разные:

```php
<?php
$user1 = new User('john', 30);
$user2 = new User('john', 25);
var_dump($user1 == $user2); // выведет false
?>
```

Давайте теперь сравним два наших объекта через ===:

```php
<?php
$user1 = new User('john', 30);
$user2 = new User('john', 30);
var_dump($user1 === $user2); // выведет false
?>
```

Чтобы две переменных с объектами действительно были равными при сравнении через ===, они должны указывать на один и тот же объект. Давайте сделаем, чтобы это было так, и сравним переменные:

```php
<?php
$user1 = new User('john', 30);
$user2 = $user1; // передача объекта по ссылке
var_dump($user1 === $user2); // выведет true
?>
```

Применение

Давайте рассмотрим применение изученной теории. Пусть у нас дан вот такой класс Employee:

```php
<?php
class Employee
{
    private $name;
    private $salary;
    public function __construct($name, $salary)
    {
        $this->name = $name;
        $this->salary = $salary;
    }
    public function getName()
    {
        return $this->name;
    }
    public function getSalary()
    {
        return $this->salary;
    }
}
?>
```

Пусть также дан такой класс EmployeesCollection для хранения коллекции работников:

```
<?php
class EmployeesCollection
{
    private $employees = []; // массив работников
    // Добавляем нового работника:
    public function add($newEmployee)
    {
        $this->employees[] = $newEmployee;
    }
    // Получаем всех работников в виде массива:
    public function get()
    {
        return $this->employees;
    }
}
?>
```

Давайте сделаем так, чтобы работник, который уже есть в нашем наборе, не добавлялся еще раз. Для этого сделаем вспомогательный метод exists, который будет принимать объект с новым работником и проверять его наличие в массиве $this->employees:

```
<?php
private function exists($newEmployee)
{
    foreach ($this->employees as $employee) {
        if ($employee == $newEmployee) { // сравниваем через ==
            return true;
        }
    }
    return false;
}
?>
```

Применим новый метод в нашем классе:

```php
<?php
class EmployeesCollection
{
    private $employees = [];
    public function add($newEmployee)
    {
        // Если такого работника нет - то добавляем:
        if (!$this->exists($newEmployee)) {
            $this->employees[] = $newEmployee;
        }
    }
    public function get()
    {
        return $this->employees;
    }
    private function exists($newEmployee)
    {
        foreach ($this->employees as $employee) {
            if ($employee == $newEmployee) {
                return true;
            }
        }
        return false;
    }
}
?>
```

Проверим работу нашего класса:

```php
<?php
$employeesCollection = new EmployeesCollection;
$employeesCollection->add(new Employee('john', 100));
$employeesCollection->add(new Employee('john', 100)); // второго
такого же не добавит
var_dump($employeesCollection->get()); // убедимся в этом
?>
```

В общем-то, мы получили устраивающий нас код. Но давайте попробуем поиграться с ним: поменяем при сравнении двойное равно на тройное:

```php
<?php
private function exists($newEmployee)
{
    foreach ($this->products as $product) {
        if ($product === $newEmployee) { // сравниваем через ===
            return true;
        }
    }
    return false;
}
?>
```

Теперь при попытке добавить нового работника с такими же значениями свойств он будет добавляться:

```
<?php
$employeesCollection = new EmployeesCollection;
$employeesCollection->add(new Employee('john', 100));
$employeesCollection->add(new Employee('john', 100)); // добавит
var_dump($employeesCollection->get());
?>
```

Но если попытаться добавить тот же объект, то добавления не произойдет:

```
<?php
$employeesCollection = new EmployeesCollection;
$employee = new Employee('john', 100);
$employeesCollection->add($employee);
$employeesCollection->add($employee); // не добавит, тк тот же объект
var_dump($employeesCollection->get());
?>
```

Функция in_array

На самом деле код метода exists можно заменить стандартной PHP функцией in_array. Нужно только знать, как она выполняет сравнение - по двойному равно или по тройному.

Из документации следует, что эта функция имеет третий необязательный параметр. Если он не установлен или равен false - функция сравнивает по двойному равно, а равен true - то по тройному.

Упростим код класса при условии сравнения объектов через двойное равно:

```
<?php
class EmployeesCollection
{
    private $employees = [];
    public function add($newEmployee)
    {
        if (!in_array($newEmployee, $this->employees, false)) {
            $this->employees[] = $newEmployee;
        }
    }
    public function get()
    {
        return $this->employees;
    }
}
?>
```

А теперь при условии сравнения на тройное равно:

```
<?php
class EmployeesCollection
{
    private $employees = [];
    public function add($newEmployee)
    {
        // Эквивалентно методу exists с ===
        if (!in_array($newEmployee, $this->employees, true)) {
            $this->employees[] = $newEmployee;
        }
    }
    public function get()
    {
        return $this->employees;
    }
}
?>
```

Определение принадлежности объекта к классу

Сейчас мы с вами изучим оператор instanceof. Данный оператор используется для определения того, является ли текущий объект экземпляром указанного класса.

Давайте посмотрим на примере. Пусть у нас даны какие-то два класса:

```
<?php
// Первый класс:
class Class1
{
}
// Второй класс:
class Class2
{
}
?>
```

Создадим объект первого класса:

```
<?php $obj = new Class1; ?>
```

Проверим принадлежность объекта из переменной $obj первому классу и второму:

```
<?php
// Выведет true, тк объект принадлежит классу Class1:
var_dump($obj instanceof Class1);
// Выведет false, тк объект НЕ принадлежит классу Class2:
var_dump($obj instanceof Class2);
?>
```

Оператор instanceof и наследование

Пусть теперь у нас есть родительский класс и дочерний:

```
<?php
// Родительский класс:
class ParentClass
{
}
// Дочерний класс:
class ChildClass extends ParentClass
{
}
?>
```

Создадим объект дочернего класса:

```
<?php $obj = new ChildClass; ?>
```

Проверим теперь с помощью instanceof, принадлежит ли наш объект классу ParentClass и классу ChildClass:

```
<?php
var_dump($obj instanceof ChildClass);  // выведет true
var_dump($obj instanceof ParentClass); // тоже выведет true
?>
```

Как вы видите из примера - оператор instanceof не делает различия при проверки между родительскими и дочерними классами.

Не путайтесь - если объект будет действительно родительского класса то, конечно же, проверка на принадлежность к дочернему классу вернет false:

```
<?php
$obj = new ParentClass; // объект родительского класса
var_dump($obj instanceof ParentClass); // выведет true
var_dump($obj instanceof ChildClass); // выведет false
?>
```

Применение

Давайте рассмотрим применение оператора instanceof на достаточно сложном примере. Пусть у нас есть вот такой класс для работников:

```
<?php
class Employee
{
    private $name;
    private $salary;
    public function __construct($name, $salary)
    {
        $this->name = $name;
        $this->salary = $salary;
    }
    public function getName()
    {
        return $this->name;
    }
    public function getSalary()
    {
        return $this->salary;
    }
}
?>
```

Пусть также есть такой класс для студентов:

```
<?php
class Student
{
    private $name;
    private $scholarship; // стипендия
    public function __construct($name, $scholarship)
    {
        $this->name = $name;
        $this->scholarship = $scholarship;
    }
    public function getName()
    {
        return $this->name;
    }
    public function getScholarship()
    {
        return $this->scholarship;
    }
}
?>
```

Как вы видите, и работник, и студент имеют имя и какой-то доход: у работника это зарплата, а у студента - стипендия.

Пусть теперь мы хотим сделать класс UsersCollection, предназначенный для хранения работников и студентов. Работников мы будем хранить в свойстве employees, а студентов - в свойстве students:

```
<?php
class UsersCollection
{
    private $employees = []; // массив работников
    private $students = []; // массив студентов
}
?>
```

Давайте теперь реализуем единый метод add для добавления и работников, и студентов. Этот метод параметром будет принимать объект и, если это работник - добавлять его в массив работников, а если студент - в массив студентов. Пример того, как мы будем пользоваться методом нашим методом после его реализации:

```
<?php
$usersCollection = new UsersCollection;
$usersCollection->add(new Employee('john', 200)); // попадет к
работникам
$usersCollection->add(new Student('eric', 100));  // попадет к
студентам
?>
```

Итак, давайте реализуем описанный метод add. Здесь нам и поможет изученный нами оператор instanceof:

```
<?php
class UsersCollection
{
    private $employees = []; // массив работников
    private $students = [];  // массив студентов
    // Добавление в массивы:
    public function add($user)
    {
        // Если передан объект класса Employee:
        if ($user instanceof Employee) {
            $this->employees[] = $user; // добавляем к работникам
        }
        // Если передан объект класса Student:
        if ($user instanceof Student) {
            $this->students[] = $user; // добавляем к студентам
        }
    }
}
?>
```

Давайте также реализуем методы для нахождения суммарной зарплаты и суммарной стипендии:

```php
<?php
class UsersCollection
{
    private $employees = [];
    private $students = [];
    public function add($user)
    {
        if ($user instanceof Employee) {
            $this->employees[] = $user;
        }
        if ($user instanceof Student) {
            $this->students[] = $user;
        }
    }
    // Получаем суммарную зарплату:
    public function getTotalSalary()
    {
        $sum = 0;
        foreach ($this->employees as $employee) {
            $sum += $employee->getSalary();
        }
        return $sum;
    }
    // Получаем суммарную стипендию:
    public function getTotalScholarship()
    {
        $sum = 0;
        foreach ($this->students as $student) {
            $sum += $student->getScholarship();
        }
        return $sum;
    }
}
?>
```

Реализуем также метод, который будет находить общую сумму платежей и работникам, и студентам:

```php
<?php
class UsersCollection
{
    private $employees = [];
    private $students = [];
    public function add($user)
    {
        if ($user instanceof Employee) {
            $this->employees[] = $user;
        }
        if ($user instanceof Student) {
            $this->students[] = $user;
        }
    }
    public function getTotalSalary()
    {
        $sum = 0;
        foreach ($this->employees as $employee) {
            $sum += $employee->getSalary();
        }
        return $sum;
    }
    public function getTotalScholarship()
    {
        $sum = 0;
        foreach ($this->students as $student) {
            $sum += $student->getScholarship();
        }
        return $sum;
    }
    // Получаем общую сумму платежей и работникам, и студентам:
    public function getTotalPayment()
    {
        return $this->getTotalScholarship() + $this->getTotalSalary();
    }
}
?>
```

Проверим работу нашего класса:

```php
<?php
$usersCollection = new UsersCollection;
$usersCollection->add(new Student('kyle', 100));
$usersCollection->add(new Student('luis', 200));
$usersCollection->add(new Employee('john', 300));
$usersCollection->add(new Employee('eric', 400));
// Получим полную сумму стипендий:
echo $usersCollection->getTotalScholarship();
// Получим полную сумму зарплат:
echo $usersCollection->getTotalSalary();
// Получим полную сумму платежей:
echo $usersCollection->getTotalPayment();
?>
```

Контроль типов при работе с объектами

Пусть у нас дан вот такой класс Employee:

```
<?php
class Employee
{
    private $name;
    private $salary;
    public function __construct($name, $salary)
    {
        $this->name = $name;
        $this->salary = $salary;
    }
    public function getName()
    {
        return $this->name;
    }
    public function getSalary()
    {
        return $this->salary;
    }
}
?>
```

Также пусть дан класс EmployeesCollection для хранения коллекции работников:

```
<?php
class EmployeesCollection
{
    private $employees = []; // массив работников
    // Добавляет работника в набор
    public function add($employee) // параметр - объект класса
Employee
    {
        $this->employees[] = $employee; // добавим объект в набор
    }
    public function getTotalSalary()
    {
        $sum = 0;
        foreach ($this->employees as $employee) {
            $sum += $employee->getSalary();
        }
        return $sum;
    }
}
?>
```

Рассмотрим внимательно метод add класса EmployeesCollection: в нем параметром передается объект класса Employee. Однако программисту, читающему наш код, сходу тяжело будет понять, что параметром метода add должен служить именно объект и именно класса Employee.

Да, мы можем оставить комментарий в нашем коде, чтобы прояснить ситуацию, но это все равно не убережет программиста от ошибок, если он попытается передать, к примеру, объект какого-нибудь другого класса или вообще массив.

Было бы круто указать тип передаваемого параметра прямо в описании функции. Ранее в учебнике мы с вами уже разбирали подобную возможность для примитивов.

Можно также явно задать и тип параметра, в который будет передаваться объект - мы можем точно сказать, объект какого класса там ожидается.

Для этого перед именем переменной параметра следует написать имя ожидаемого класса, в нашем случае Employee.

Давайте переделаем наш метод add:

```
<?php
class EmployeeCollection
{
    private $employees = [];
    // Явно укажем тип параметра:
    public function add(Employee $employee)
    {
        $this->employees[] = $employee;
    }
    public function getTotalSalary()
    {
        $sum = 0;
        foreach ($this->employees as $employee) {
            $sum += $employee->getSalary();
        }
        return $sum;
    }
}
?>
```

Статические методы в ООП на PHP

При работе с классами можно делать методы, которые для своего вызова не требуют создания объекта. Такие методы называются статическими. Чтобы объявить метод статическим, нужно после модификатора доступа написать ключевое слово static:

```
<?php
class Test
{
    // Статический метод:
    public static function method()
    {
        return '!!!';
    }
}
?>
```

Чтобы обратиться к статическому методу, нужно написать имя класса, потом два двоеточия и имя метода, объект класса при этом создавать не надо, вот так:

```
<?php
echo Test::method(); // выведет '!!!'
?>
```

Пример

Давайте рассмотрим статические методы на более практическом примере. Пусть у нас дан вот такой математический класс Math (пока без статических методов):

```
<?php
class Math
{
    // Находит сумму:
    public function getSum($a, $b)
    {
        return $a + $b;
    }
    // Находит произведение:
    public function getProduct($a, $b)
    {
        return $a * $b;
    }
}
?>
```

Давайте воспользуемся нашим классом:

```
<?php
$math = new Math; // создаем объект класса
echo $math->getSum(1, 2) + $math->getProduct(3, 4); // используем
методы
?>
```

Наш класс Math представляет собой просто набор методов и, фактически, нам нужен только один объект этого класса. В таком случае удобно объявить методы класса статическими и вообще не создавать объект этого класса, а сразу использовать его методы. Сделаем это:

```php
<?php
class Math
{
    public static function getSum($a, $b)
    {
        return $a + $b;
    }
    public static function getProduct($a, $b)
    {
        return $a * $b;
    }
}
?>
```

Воспользуемся методами нашего класса без создания объекта класса:

```php
<?php
echo Math::getSum(1, 2) + Math::getProduct(3, 4);
?>
```

Статические методы внутри класса

Если вы хотите использовать статические методы внутри класса, то к ним следует обращаться не через $this->, а с помощью self::.

Для примера добавим в наш класс Math метод getDoubleSum, который будет находить удвоенную сумму чисел. Используем внутри нового метода уже существующий метод getSum:

```php
<?php
class Math
{
    // Найдем удвоенную сумму:
    public static function getDoubleSum($a, $b)
    {
        return 2 * self::getSum($a, $b); // используем другой метод
    }
    public static function getSum($a, $b)
    {
        return $a + $b;
    }
    public static function getProduct($a, $b)
    {
        return $a * $b;
    }
}
?>
```

Воспользуемся новым методом:

```php
<?php
echo Math::getDoubleSum(1, 2);
?>
```

Практика

Пусть у нас дан вот такой класс ArraySumHelper, который мы рассматривали в одном из предыдущих уроков:

```php
<?php
class ArraySumHelper
{
    public function getSum1($arr)
    {
        return $this->getSum($arr, 1);
    }
    public function getSum2($arr)
    {
        return $this->getSum($arr, 2);
    }
    public function getSum3($arr)
    {
        return $this->getSum($arr, 3);
    }
    public function getSum4($arr)
    {
        return $this->getSum($arr, 4);
    }
    private function getSum($arr, $power) {
        $sum = 0;
        foreach ($arr as $elem) {
            $sum += pow($elem, $power);
        }
        return $sum;
    }
}
?>
```

Статические свойства в ООП на PHP

Кроме статических методов можно также делать и статические свойства. Такие свойства также объявляются с помощью ключевого слова static:

```php
<?php
class Test
{
    public static $property; // статическое свойство
}
?>
```

Можно что-то записать в статическое свойство и прочитать из него. При этом имя свойства указывается вместе с долларом:

```php
<?php
Test::$property = 'test';
echo Test::$property; // выведет 'test'
?>
```

Статическое свойство внутри класса

Можно также обращаться к статическим свойствам внутри самого класса, используя self::, смотрите пример:

```php
<?php
class Test
{
    // Приватное статическое свойство:
    private static $property;
    // Статический метод для задания значения свойства:
    public static function setProperty($value)
    {
        self::$property = $value; // записываем данные в наше static
свойство
    }
    // Статический метод для получения значения свойства:
    public static function getProperty()
    {
        return self::$property; // прочитываем записанные данные
    }
}
?>
```

Воспользуемся нашим классом:

```php
<?php
Test::setProperty('test'); // запишем данные в свойство
echo Test::getProperty(); // выведем на экран
?>
```

Можно также задать начальное значение свойства:

```php
<?php
class Test
{
    // Начальное значение свойства:
    private static $property = 'test';
    public static function getProperty()
    {
        return self::$property;
    }
}
echo Test::getProperty(); // выведет 'test'
?>
```

Применение

Пусть у нас есть класс Geometry для работы с геометрическими фигурами. В этом классе есть методы для определения площади круга и длины окружности:

```
<?php
class Geometry
{
    // Площадь круга:
    public static function getCircleSquare($radius)
    {
        return 3.14 * $radius * $radius;
    }
    // Длина окружности:
    public static function getCirclecircuit($radius)
    {
        return 2 * 3.14 * $radius;
    }
}
?>
```

Как вы видите, в обоих методах используется число Пи, равное 3.14. Было бы удобно вынести это число в статическое свойство класса - в этом случае значение числа Пи будет задаваться в одном месте и мы легко сможем поменять его в случае необходимости (например, написать больше знаков в дробной части).

Давайте сделаем это:

```
<?php
class Geometry
{
    private static $pi = 3.14; // вынесем Пи в свойство
    public static function getCircleSquare($radius)
    {
        return self::$pi * $radius * $radius;
    }
    public static function getCirclecircuit($radius)
    {
        return 2 * self::$pi * $radius;
    }
}
?>
```

Объект со статическими свойствами и методами

Вы уже знаете, что статические свойства и методы можно использовать, не создавая объект класса. На самом деле, однако, класс может содержать как статические свойства и методы, так и обычные.

Давайте посмотрим, как с этим работать и какие преимущества это дает. Пусть у нас есть класс Test одновременно и со статическим свойством, и с обычным:

```
<?php
class Test
{
    public static $staticProperty; // публичное статическое свойство
    public $usualProperty; // обычное свойство
}
?>
```

Давайте, к примеру, поработаем с его обычным свойством:

```
<?php
$test = new Test; // создаем объект класса
$test->usualProperty = 'usual'; // записываем значение
echo $test->usualProperty; // выведет 'usual'
?>
```

А теперь используем статическое свойство, не создавая объект этого класса:

```
<?php
Test::$staticProperty = 'static'; // записываем значение
echo Test::$staticProperty; // выведет 'static'
?>
```

На самом деле, если у нас есть переменная с объектом класса, то у этой переменной также будет доступно статическое свойство:

```
<?php
$test = new Test; // создаем объект класса
$test::$staticProperty = 'static'; // записываем значение
echo $test::$staticProperty; // выведет 'static'
?>
```

Разницы нет - мы к одному и тому же статическому свойству можем обращаться и так, и так. Вот пример, иллюстрирующий это:

```
<?php
// Записываем значение еще ДО создания объекта:
Test::$staticProperty = 'static';
// Создаем объект класса:
$test = new Test;
// Выводим статическое свойство:
echo $test::$staticProperty; // выведет 'static'
?>
```

Вот еще пример:

```
<?php
// Создаем объект класса:
$test = new Test;
// Записываем значение в статическое свойство:
$test::$staticProperty  = 'static';
// Выводим значение, обратившись к классу:
echo Test::$staticProperty; // выведет 'static'
// Выводим значение, обратившись к объекту класса:
echo $test::$staticProperty; // выведет 'static'
?>
```

Несколько объектов

Статические свойства принадлежат не какому-то объекту класса, а самому классу, хотя объекты класса и имеют доступ к этим свойствам.

На практике это означает то, что если у нас есть несколько объектов класса - статические свойства у них будут общие. То есть, если в одном объекте поменять значение статического свойства - изменения произойдут во всех объектах.

Давайте посмотрим на примере:

```
<?php
$test1 = new Test; // первый объект
$test2 = new Test; // второй объект
$test1::$staticProperty = 'static'; // запишем значение, используя
первый объект
echo $test1::$staticProperty; // выведет 'static'
echo $test2::$staticProperty; // также выведет 'static'
?>
```

Статические методы и $this

Пусть у нас есть класс Test с двумя свойствами, статическим и обычным:

```
<?php
class Test
{
    public static $staticProperty = 'static'; // статическое свойство
    public $usualProperty = 'usual'; // обычное свойство
}
?>
```

Давайте выведем значения этих свойств в обычном методе method:

```
<?php
class Test
{
    public static $staticProperty = 'static'; // статическое свойство
    public $usualProperty = 'usual'; // обычное свойство
    // Обычный метод:
    public function method()
    {
        var_dump(self::$staticProperty); // выведет 'static'
        var_dump($this->usualProperty);  // выведет 'usual'
    }
}
$test = new Test;
$test->method(); // обычный метод - вызываем через ->
?>
```

Из примера видно, что в обычном методе доступны как статические, так и обычные свойства (и методы). Пусть теперь наш метод method будет статическим. В этом случае он сможет обратиться с статическим методом и свойствам, но к обычным - нет.

Почему: потому что внутри статических методов недоступен $this. Это происходит из-за того, что статические методы могут вызываться вне контекста объекта, просто обращаясь к имени класса.

А ведь $this внутри класса как раз-таки ссылается на объект этого класса. Нет объекта - $this ни на что не ссылается. Убедимся в этом: переделаем наш метод на статический - теперь обращение к обычному свойству внутри нашего метода будет выдавать ошибку:

```
<?php
class Test
{
    public static $staticProperty = 'static'; // статическое свойство
    public $usualProperty = 'usual'; // обычное свойство
    // Переделали на статический метод:
    public static function method()
    {
        var_dump(self::$staticProperty); // выведет 'static'
        var_dump($this->usualProperty);  // выдаст ошибку
    }
}
$test = new Test;
$test::method(); // статический метод - вызываем через ::
?>
```

Применение

Пусть у нас есть вот такой класс User:

```
<?php
class User
{
    public $name;
    public function __construct($name)
    {
        $this->name = $name;
    }
}
?>
```

Давайте сделаем так, чтобы этот класс подсчитывал количество своих объектов. Для этого сделаем статическое свойство count. Изначально запишем в него значение 0, а при создании каждого нового объекта будем увеличивать это значение на 1.

Будем увеличивать значение нашего счетчика в конструкторе объекта:

```php
<?php
class User
{
    public static $count = 0; // счетчик объектов
    public $name;
    public function __construct($name)
    {
        $this->name = $name;
        // Увеличиваем счетчик при создании объекта:
        self::$count++;
    }
}
?>
```

Проверим, что все работает:

```php
<?php
$user1 = new User('user1'); // создаем первый объект класса
echo User::$count; //выведет 1
$user2 = new User('user2'); // создаем второй объект класса
echo User::$count; //выведет 2
?>
```

Улучшим наш код

Не очень хорошо то, что наш счетчик публичный - его случайно можно изменить снаружи класса. Давайте сделаем его доступным только для чтения. Для этого объявим его приватным и сделаем для него статический метод-геттер getCount:

```php
<?php
class User
{
    private static $count = 0;
    public $name;
    public function __construct($name)
    {
        $this->name = $name;
        // Увеличиваем счетчик при создании объекта:
        self::$count++;
    }
    // Метод, возвращающий значение счетчика:
    public static function getCount()
    {
        // Выводим значение счетчика:
        return self::$count;
    }
}
?>
```

Проверим:

```
<?php
$user1 = new User('user1'); // создаем первый объект класса
echo User::getCount(); //выведет 1
$user2 = new User('user2'); // создаем второй объект класса
echo User::getCount(); //выведет 2
?>
```

Константы классов в ООП на PHP

Сейчас мы с вами разберем константы классов. Константы по сути являются свойствами, значения которых нельзя изменить. Неизменяемые свойства нужны для того, чтобы хранить какие-то значения, которые являются постоянными и не должны быть случайно изменены.

Чтобы создать константу, ее нужно объявить через ключевое слово const, затем написать имя константы без доллара и обязательно сразу же задать ее значение:

```
<?php
class Test
{
    const constant = 'test'; // задаем константу
}
?>
```

Общепринято имена констант писать большими буквами, то есть не constant, а CONSTANT. Это делается для того, чтобы визуально легко было отличать константы в коде.

Давайте поправим наш класс:

```
<?php
class Test
{
    // Задаем константу:
    const CONSTANT = 'test';
}
?>
```

Давайте теперь рассмотрим, как прочитать значения константы. Здесь следует сказать то, что константы класса больше похожи не на обычные свойства, а на статические.

Это значит, что константы класса задаются один раз для всего класса, а не отдельно для каждого объекта этого класса.

Поэтому обращение к константам происходит почти так же, как и для статических свойств: пишем имя класса, два двоеточия и название константы без доллара перед именем:

```
<?php
echo Test::CONSTANT; // выведет 'test'
?>
```

Как уже упоминалось выше, значения констант можно прочитывать, но не записывать. Попытка что-то записать в нее выдаст ошибку:

```
<?php
Test::CONSTANT = 'test'; // выдаст ошибку
?>
```

Обращение к константам внутри класса

Внутри класса также можно обратится к константе через ::self, вот так:

```
<?php
class Test
{
    const CONSTANT = 'test';
    function getConstant() {
        return self::CONSTANT;
    }
}
?>
```

Воспользуемся нашим методом:

```
<?php
$test = new Test;
echo $test->getConstant(); // выведет 'test'
?>
```

Применение

Давайте сделаем класс Circle, с помощью которого можно будет найти площадь круга и длину окружности. Для работы с кругом нам понадобится число Пи, равное 3.14. Логично будет для хранения этого числа использовать константу, чтобы случайно где-нибудь в коде наше число Пи вдруг не поменялось.

Вот частичная реализация нашего класса:

```php
<?php
class Circle
{
    const PI = 3.14; // запишем число ПИ в константу
    private $radius; // радиус круга
    public function __construct($radius)
    {
        $this->radius = $radius;
    }
    // Найдем площадь:
    public function getSquare()
    {
        // Пи умножить на квадрат радиуса
    }
    // Найдем длину окружности:
    public function getCircuit()
    {
        // 2 Пи умножить на радиус
    }
}
?>
```