

# КЛАССЫ И ОБЪЕКТЫ



ЕВГЕНИЙ КОРЫТОВ / РУКОВОДИТЕЛЬ ГРУППЫ ВЕБ-РАЗРАБОТКИ, CONDÉ NAST



# ЕВГЕНИЙ КОРЫТОВ

Руководитель группы веб-разработки,  
Condé Nast



korytoff@gmail.com



korytoff



# ПРИВЕТ

- В веб-разработке с 2006 года
- Работал в Unigine, WinNER
- Делал десктопные приложения на PHP + XUL
- 3 года преподавания в ВУЗе
- ...но эта лекция далась действительно сложно)



# КАК ПРЕДЛАГАЕТСЯ РАБОТАТЬ?

- Каждую лекцию вспоминаем, что прошли на предыдущей
- Вопросы - категорически приветствуются
- Будем пытаться выработать понимание, а не учить теорию
- Буду благодарен за любые отзывы после лекции
- Без практики - любой материал бесполезен



## ЧТО МЫ УЖЕ ДОЛЖНЫ ЗНАТЬ?

- Ассоциативные массивы
- Функции и параметры по умолчанию
- Стандартные типы данных
- Базовые конструкции языка (циклы, if, switch...)
- Области видимости переменных
- Передача переменных по ссылке

---

# ОКЕЙ, ЧТО ДАЛЬШЕ?

1. Много новой информации, от которой будет болеть голова
2. Классы
3. Свойства и методы
4. Объекты (экземпляры классов)
5. `$this` - это...
6. Магические методы. Конструктор.
7. Области видимости свойств и методов
8. Статические свойства и методы
9. Константы классов



# ЗАЧЕМ УЧИТЬ ООП?

# ПРОГУЛКА ПО ИСТОРИИ

1. Императивная парадигма
2. Процедурное программирование
3. Объектно-ориентированное программирование

В общем и целом, цель каждого подхода - упрощать работу с более сложными системами.

Вывод?

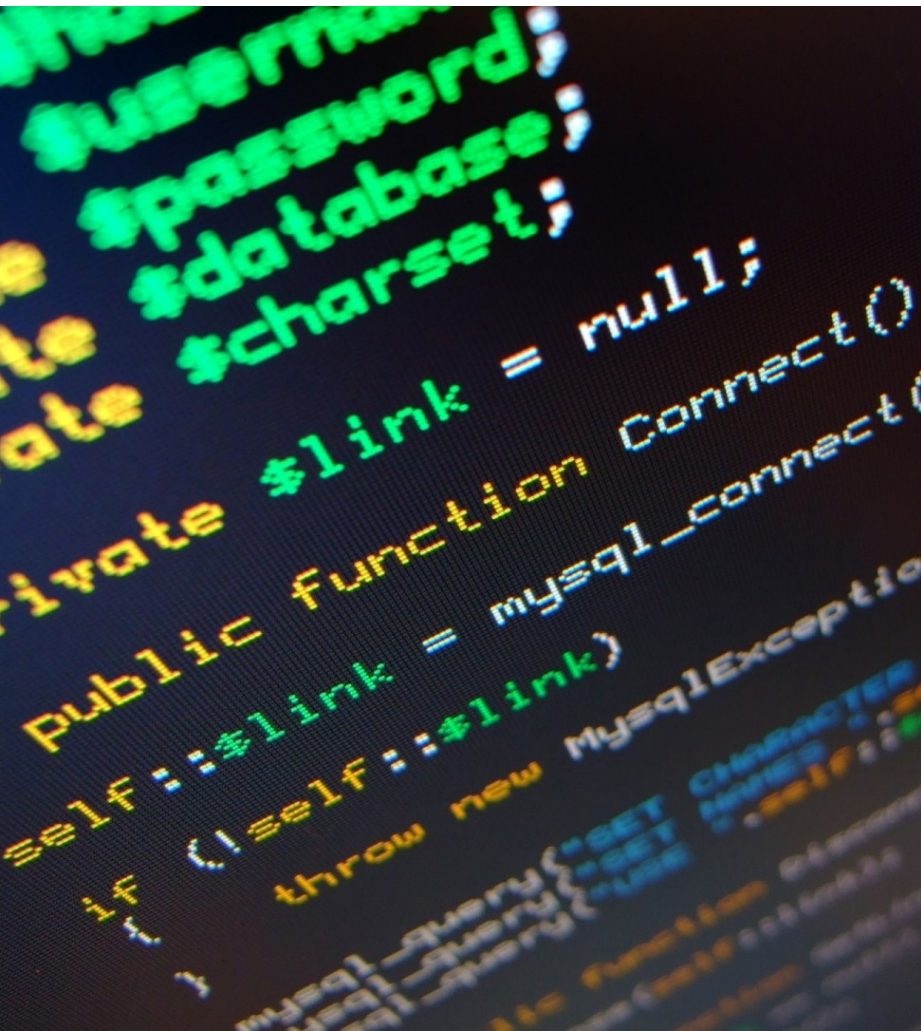


# ТО, ЧТО ВАМ НЕ РАССКАЖЕТ НИ ОДИН ЧЕЛОВЕК, КОТОРЫЙ БУДЕТ УЧИТЬ ВАС ООП

Всё можно сделать и без использования объектов

НО

1. ОО-подход - это сейчас стандарт
2. Меньше шансов сделать не так, как задумывалось
3. Более читабельный код
4. Любая сложная система без объектов = спагетти-код



---

## ТРИ ПРИНЦИПА ООП

- Инкапсуляция
- Полиморфизм
- Наследование

---

**ПОПРОБУЕМ САМИ  
ПОНЯТЬ, ЧТО ТАКОЕ  
ИНКАПСУЛЯЦИЯ**



# ЗАДАЧА

Нас попросили доработать небольшой Интернет-магазин и на главной странице вывести 10 самых популярных товаров.

Нам всё равно, где они хранятся, но мы можем выгрузить их в массив, чтобы с ним работать

Каждый товар имеет цену и название.

Итак - есть массив товаров, нужно его вывести. Вроде, всё не слишком сложно, верно?

# ПЛЁВОЕ ДЕЛО!

```
1 $products = [ //массив, которые мы получили
2     ['name' => 'Булгаков. Мастер и Маргарита', 'price' =>
3     ['name' => 'Samsung Galaxy S7 Edge', 'price' => 50000
4     ['name' => 'Apple iPhone 7', 'price' => 150],
5         //любое иное количество товаров
6 ];
7
8 foreach ($products as $product) { //выводим
9     echo $product['name'] . ' стоит ' . $product['price']
10 }
```



## "ДАВАЙТЕ ПОИГРАЕМСЯ С ЦВЕТАМИ"

Всё классно, заказчик доволен. Но единственное, что в бизнесе постоянно - это необходимость регулярно что-то менять.

Внезапно было решено устроить распродажу и дать на все смартфоны скидку 10 процентов.

Что же, для этого массив должен содержать категории и нам нужно добавить подсчет скидки.

Легче лёгкого!

# ЛЕГЧЕ ЛЁГКОГО!

```
1  $products = [ //массив, которые мы получили
2      ['name' => 'Булгаков. Мастер и Маргарита',
3          'category' => 'Книги', 'price' => 100],
4      ['name' => 'Samsung Galaxy S7 Edge',
5          'category' => 'Смартфон', 'price' => 50000],
6      //любое иное количество товаров
7  ];
```

## ЛЕГЧЕ ЛЁГКОГО! (ВЫВОДИМ)

```
1  foreach ($products as $product) { //ВЫВОДИМ
2      echo $product['name'] . ' стоит '
3      if ($product['category'] != 'Смартфон') {
4          echo $product['price'];
5      } else {
6          echo round($product['price'] - $product['price'])
7      }
8  }
```



---

## ЧТО ЕСЛИ Я СКАЖУ ТЕБЕ, ЧТО СКИДКА МОЖЕТ БЫТЬ У КАЖДОГО ТОВАРА СВОЯ?

А тем временем заказчик не успокаивается. Ему всё нравится, но есть нюанс - у каждого товара может быть задана своя скидка. И выводить нужно минимальную скидку.

Ох уж эти заказчики. Ладно. И так - у нас есть 2 скидки. Одна в связи с акцией, вторая в связи со **свойством** товара.

Что же, сделаем. Но мы понимаем, что проще будет эту логику заложить в функцию и использовать её везде где нужен вывод цены (например, в корзине).

## ДОБАВЛЯЕМ СКИДКУ К ТОВАРУ

```
1  $products = [ //массив, которые мы получили
2      ['name' => 'Булгаков. Мастер и Маргарита', 'category'
3          'discount' => 5, 'price' => 100],
4      ['name' => 'Samsung Galaxy S7 Edge', 'category' => 'C
5          'discount' => 5, 'price' => 50000],
6          //любое иное количество товаров
7  ];
```

## И ПИШЕМ ФУНКЦИЮ ДЛЯ РАССЧЕТА ИТОГОВОЙ ЦЕНЫ

```
1 function getPrice($price, $discount, $category)
2 {
3     $phoneDiscount = ($category == 'Смартфон') ? 10 : 0;
4     $discount = min($discount, $phoneDiscount);
5
6     if ($discount) {
7         return round($price - ($price * $discount / 100));
8     } else {
9         return $price;
10 }
```

## А ТЕПЕРЬ ВЫВОДИМ

```
1  foreach ($products as $product) { //выводим
2      echo $product['name'] . ' стоит '
3          . getPrice($product['price'], $product['discount']
4      }
```



## А ТЕПЕРЬ ЗАКАЗЧИК ХОЧЕТ...

Ладно, ладно. Пока ничего не хочет. Заказчик доволен и дал нам день передохнуть. Но завтра обещал придумать ещё что-то новенькое.

Давайте пока взглянем свежим взглядом на то, что есть.

---

## ЧТО МЫ ИМЕЕМ?

1. Функцию, которая абсолютно бесполезна в любом другом контексте. Она не сможет работать не с продуктом.
2. Новый разработчик, придя в проект понятия не имеет, что цену надо пользоваться этой функцией для вывода цены
3. Мы работаем с индексами и завязаны на них. Если их не окажется или опечатаемся - возникнет ошибку, которую сложно отловить
4. Если, например, нам нужно будет рассчитать цену товара в зависимости от его веса - мы напишем еще одну функцию, о которой нужно не забыть, и которая бесполезна для чего-то еще
5. Если кто-то в коде случайно заменит цену, можем ли мы ему как-то помешать?


---

# КАК СДЕЛАТЬ ПРАВИЛЬНО И ЛУЧШЕ?

Да, да, мы наконец перейдём к тому, зачем все собрались. К объектам.

Они позволят нам собрать в одном месте всю логику, которая относится к одному контексту. В нашем случае - к продукту.

Это и есть первый и базовый принцип ООП - **Инкапсуляция**



# **БАЗОВЫЕ ПОНЯТИЯ И КОНСТРУКЦИИ ООП. КЛАССЫ.**



# ВСЁ НАЧИНАЕТСЯ С КЛАССА

Класс - это некий тип / структура / шаблон на базе которого мы будем создавать объекты.

По сути, основная задача классов - задать контекст, собрать в одном месте всё, что относится к одной сущности.

```
1  class SimpleClass
2  {
3      // пустой класс
4      // с ним тоже можно уже работать
5  }
```

# СВОЙСТВА

Класс может содержать свойства и методы. Разберемся со свойствами.

```
1  class SimpleClass
2  {
3      public $simpleProperty; //свойство без значения
4      var $anotherSimpleProperty; // var == public, но испо
5
6      // свойство со значением по умолчанию
7      public $propertyWithDefaultValue = 100;
8  }
```

# МЕТОДЫ

Теперь разберемся с методами

```
1  class SimpleClass
2  {
3      public function simpleMethod()
4      {
5          //тело метода - тут может быть любая логика
6      }
7  }
```

## МЕТОДЫ (Ч.2)

Методы, как и функции, могут принимать входные параметры, иметь параметры по умолчанию, возвращать или не возвращать значения.

```
1 class SimpleClass
2 {
3     public function anotherMethod($a, &$b, $c = 'Значение')
4     {
5         //тело метода - тут может быть любая логика
6     }
7 }
```

## И ВСЁ ВМЕСТЕ

```
1  class SimpleClass
2  {
3      public $simpleProperty; //свойство
4      public $simplePropertyWithValue = 100; //свойство
5
6      public function simpleMethod() {} //пустой метод без
7      public function anotherMethod($a, &$b, $c = 'Значение
8      {
9          //тело метода - тут может быть любая логика
10     }
11 }
```

## КАК ВЫГЛЯДЕЛ БЫ КЛАСС ДЛЯ НАШЕЙ ЗАДАЧИ ВЫШЕ?


```
1  class Product
2  {
3      public $name; //свойство
4      public $category; //свойство
5      public $price; //свойство
6      public $discount; //свойство
7
8      public function getPrice() //метод
9      {
10         // тут базовая логика из нашей функции
```

---

## КАК С ЭТИМ РАБОТАТЬ?

Окей, вот мы описали некоторую структуру. А что дальше? Как с ней работать?

Что же, а теперь мы можем создать **любое количество объектов** на базе этой структуры.



# **БАЗОВЫЕ ПОНЯТИЯ И КОНСТРУКЦИИ ООП. ОБЪЕКТЫ.**



# КАК СОЗДАВАТЬ ОБЪЕКТЫ

Объекты - это конкретные и уникальные экземпляры классов

```
1 class SimpleClass //это класс
2 {
3
4 }
5
6 $simpleObject = new SimpleClass(); //а вот это объект
7 $anotherObject = new SimpleClass(); //и вот это объект
8 $whatAboutThat = new SimpleClass(); //и даже это объект
```

# РАБОТА СО СВОЙСТВАМИ И МЕТОДАМИ

Итак, еще раз. Допустим у нас есть класс со свойствами и методами. Как же с ними работать?

```
1 class SimpleClass
2 {
3     public $simpleProperty; //свойство
4     public $simplePropertyWithValue = 200; //свойство
5
6     public function simpleMethod() {
7         return 'Результат выполнения метода';
8     }
9 }
```

## РАБОТА СО СВОЙСТВАМИ И МЕТОДАМИ (Ч. 2)

Алгоритм такой: создаём объект и обращаемся к его свойствам и методам

```
1 $object = new SimpleClass();
2 $object->simpleProperty = 100; //присваиваем значение сво
3
4 echo $object->simpleProperty; //считываем свойство и выво
5 echo $object->simplePropertyWithValue; //считываем свойс
6
7 //обращаемся к методу и выводим его результат
8 echo $object->simpleMethod(); //что будет выведено?
```

## РАБОТА СО СВОЙСТВАМИ И МЕТОДАМИ (Ч. 3)

Важно понять - у каждого объекта свои свойства и методы, класс это тип.

```
1  $object = new SimpleClass();
2  $anotherObject = new SimpleClass();
3
4  $object->simplePropertyWithValue = 1;
5  echo $anotherObject->simplePropertyWithValue; //что выведет?
6  echo $object->simplePropertyWithValue; //что выведет?
```

## НА ПРИМЕРЕ ПРОДУКТОВ

Вспомним, какой класс у нас был для описания продуктов.

```
1  class Product
2  {
3      public $name; //свойство
4      public $category; //свойство
5      public $price; //свойство
6      public $discount; //свойство
7
8      public function getPrice() { // тут базовая логика из
9  }
```

## НА ПРИМЕРЕ ПРОДУКТОВ (Ч.2)

И вот как мы с ними могли бы работать

```
1 $product = new Product();  
2 $product->name = 'Apple iPhone 7';  
3 $product->price = 50000;  
4  
5 $anotherProduct = new Product();  
6 $anotherProduct->name = 'Samsung Galaxy S7 Edge';  
7 $anotherProduct->price = 50000;
```



Чтобы лучше понять разницу между классами и объектами, представьте, что формочка для печенек слева, определяющая их форму - это **Класс**, а сами печеньки которые в итоге получаются - это **Объекты** . :)

## ПРО ПЕЧЕНЬКИ (Ч.2)

Таким образом, в момент создания объекта мы примеряем формочку и берем некоторые стартовые свойства и методы объекта. Потом мы вольны их менять (никто нам не запрещает скатать шарик из медвежонка)

Свойства - это некоторые параметры объекта. Например, название, материал, и т.д.

Методы, как правило, - это некоторые действия, которые можно с объектом делать.

Обычно, именно по этим принципам и строятся классы и объекты.



---

# ВЕРНЕМСЯ В РЕАЛЬНЫЙ МИР

В связи со всем вышесказанным, часто, для понимания ООП рекомендуют рассматривать объекты реального мира. Примеры:

1. Машина - класс, каждая Audi Q7, которую вы видите на улице - конкретный объект класса "Машина"
2. Здание - класс, дом в котором вы живете - объект класса "Здание"
3. Ноутбук - класс, конкретный ноутбук ASUS по которому я рассказываю эту лекцию - объект.
4. Здесь может быть ваш пример

## ПОПРОБУЕМ ПОРАЗМЫСЛИТЬ

На самом деле, из того, что мы уже успели рассмотреть, никакой разницы с ассоциативными массивами пока нет. По сути, мы стали просто вместо индексов использовать свойства.

```
1 $product = new Product();  
2 $product->name = 'Apple iPhone 7';  
3 $product->price = 50000;  
4 // можно сказать, почти идентично  
5 $productArray['name'] = 'Apple iPhone 7';  
6 $productPrice['price'] = 50000;
```

Но это только начало :)

## ПОМНИТЕ НАШУ ФУНКЦИЮ ДЛЯ РАССЧЕТА ЦЕНЫ?

```
1 function getPrice($price, $discount, $category)
2 {
3     $phoneDiscount = ($category == 'Смартфон') ? 10 : 0;
4     $discount = min($discount, $phoneDiscount);
5
6     if ($discount) {
7         return round($price - ($price * $discount / 100));
8     } else {
9         return $price;
10 }
```

## А МЕТОД?

Я намеренно не передавал в неё никаких параметров. Почему? Потому что сама идея инкапсуляции была бы бессмысленна, если бы мы не имели возможность работать со свойствами структуры внутри неё самой.

```
1 class Product
2 {
3     //тут находятся свойства
4
5     public function getPrice() { // тут базовая логика из
6 }
```



**\$THIS - ЭТО...**

# УТВЕРЖДЕНИЕ

Итак. Мы должны иметь возможность работать со свойствами и методами внутри контекста. Как это сделать? Встречаем - \$this

```
1 class SimpleClass
2 {
3     public $simpleProperty = 100;
4
5     public function simpleMethod()
6     {
7         $this->simpleProperty = 200; //меняем значение с
8     }
9 }
```

# УКАЗАТЕЛЬ НА ОБЪЕКТ

Важно: `$this` указывает на конкретный объект, в рамках которого вызывается.

```
1  $object = new SimpleClass();
2  $anotherObject = new SimpleClass();
3
4  echo $object->simpleProperty; // 100
5  echo $anotherObject->simpleProperty; // 100
6  $object->simpleMethod(); // поменяли значение свойства В
7  echo $object->simpleProperty; // стало 200
8  echo $anotherObject->simpleProperty; // осталось 100!
```

## ЕЩЁ ДЛЯ ПОНИМАНИЯ

Допустим, есть класс Car с методом changeColor.

```
1  class Car
2  {
3      //тут иные свойства и методы
4      public $color = 'Белая'; // все машины по умолчанию -
5
6      public function changeColor($color) //метод перекраши
7      {
8          //заменяем цвет конкретного объекта
9          $this->color = $color; //на тот, что пришел в мет
10     }
11 }
```



## ЕЩЁ ДЛЯ ПОНИМАНИЯ Ч.2

От того, что мы перекрасили Ауди в красный, BMW ни тепло, ни холодно. Она как была белой, так ей и остаётся.

```
1 $audi = new Car();
2 $bmw = new Car();
3
4 $audi->changeColor( 'Красный' ); //теперь мы имеем красную
5 echo $bmw->color; //БМВ остается белым, его никто не пере
6
7 //это ещё одна Ауди
8 $anotherAudi = new Car(); // какого она цвета?
```

## КАК ЭТО ПРИМЕНИТЬ К ПРОДУКТАМ?

Давайте теперь перенесём нашу функцию, которой мы передавали три параметра в метод (немного упростим) и посмотрим, как это в итоге будет работать

```
1  class Product
2  {
3      public $name;
4      public $price;
5      public $discount;
6      public $category;
7
8      public function getPrice() { //это будет на следующем
9  }
```

## КОД МЕТОДА

```
1  class Product
2  { //тут свойства - name, category, discount, price
3      public function getPrice()
4      {
5          // упростил - нет скидки за смартфоны
6          if ( $this->discount ) {
7              return round( $this->price - ( $this->price *
8          } else {
9              return $this->price;
10     }
11 }
```

## КОД МЕТОДА (Ч.2)

```
1  class Product
2  { //тут свойства - name, category, discount, price
3      public function getPrice()
4      {
5          $phoneDiscount = ($this->category == 'Смартфон'
6          $discount = min($this->discount, $phoneDiscount)
7
8          if ($this->discount) {
9              return round($this->price - ($this->price *
10          } else {
11              return $this->price;
12  }
```

## КАК С ЭТИМ ЖИТЬ?

```
1  $product = new Product;  
2  $product->name = 'Apple iPhone 7';  
3  $product->price = 50000;  
4  $product->discount = 10; //в процентах  
5  echo $product->getPrice(); //выведет 45000;  
6  
7  $anotherProduct = new Product;  
8  $anotherProduct->name = 'Samsung Galaxy S4';  
9  $anotherProduct->price = 50000;  
10 echo $anotherProduct->getPrice(); //сколько выведет?
```

# ОБДУМАЕМ ЕЩЁ РАЗ

Чего нам удалось добиться?

1. Всё в рамках контекста - продукта - собрано в одном месте (инкапсуляция). Это банально удобно (не нужно бегать по коду и собирать что к чему относится)
2. Методу не нужно передавать параметры, он осведомлен о свойствах и методах объекта, который его вызывает (и это тоже инкапсуляция)
3. В результате, код более читабельный, с ним удобней работать, меньше шансов ошибиться (как, например, сделать опечатку в индексе массива)

## ОБДУМАЕМ ЕЩЁ РАЗ (Ч.2)

Объективно остаётся две проблемы

1. Задавать свойство каждому объекту в отдельной строке - банально неудобно, особенно, если основных свойств много.
2. Никто не мешает не пользоваться методом `getPrice` и выводить цену просто, как `$product->price`, игнорируя скидку

К счастью, и эти проблемы можно решить в рамках объектов. (А вот в рамках массивов - это никак не решить)

Для начала добавим немного магии. )



# МАГИЧЕСКИЕ МЕТОДЫ. КОНСТРУКТОР.



# О МАГИЧЕСКИХ МЕТОДАХ

Магических методов в php довольно много - [ссылка на документацию](#)

Все они - зарезервированные слова и начинаются с двух знаков подчеркивания \_\_. Мы рассмотрим их детальней на следующих лекциях.

А сейчас научимся работать с самым популярным из магических методов - конструктором.

Готовы к магии?

# КОНСТРУКТОР

Конструктор - это специальный метод, который вызывается в момент создания объекта

```
1 class SimpleClass
2 {
3     public function __construct()
4     {
5         //содержимое конструктора
6         echo 'Был создан объект класса SimpleClass';
7     }
8 }
```

## КОНСТРУКТОР 4.2

Для класса выше, картина будет такая:

```
1  $object = new SimpleClass(); // вот в этот момент вызывается конструктор
2  $anotherObject = new SimpleClass(); // и тут вызывается конструктор
3  $superObject = new SimpleClass(); // и вот тут
4
5  // Сколько строк "Был создан объект класса SimpleClass"
6  // будет выведено в данном примере?
```

## КОНСТРУКТОР 4.3

В конструктор можно также передавать параметры, как и в функцию. И обычно конструктор используется для установки некоторых базовых значений для **объекта**.

```
1 class SimpleClass
2 {
3     public $property;
4     public function __construct($property)
5     {
6         $this->property = $property;
7     }
8 }
```

## КОНСТРУКТОР Ч.4

Вот как с этим работать

```
1 //вот как передавать параметры сразу при создании объекта
2 $object = new SimpleClass(100); //конструктор
3 $anotherObject = new SimpleClass(200); //конструктор
4
5 echo $object->property; // что выведет?
6 echo $anotherObject->property; // что выведет?
7
8 $object->property = 1000;
9 echo $object->property; //что выведет?
```



## ПОЧЕМУ ЭТО ХОРОШО

1. В ряде случаев объекты бессмысленны и не могут работать без задания соответствующих параметров. Например, товар. Как минимум, мы изначально, при создании объекта должны знать что мы продаем (название) и цену. Если мы забудем что-то из этого - всё будет плохо.
2. Меньше кода пишем, в момент создания объекта мы понимаем, что мы создаём и зачем. Очень удобно.

## НА ПРИМЕРЕ С ПРОДУКТОМ

```
1  class Product
2  {
3      //тут свойства name, price, discount, category
4      public function __construct($name, $price, $discount
5      {
6          $this->name = $name;
7          $this->price = $price;
8          $this->discount = $discount;
9      }
10     //тут метод getPrice
11 }
```

## КАК С ЭТИМ РАБОТАТЬ?

```
1 // согласитесь - понятней и удобней
2 $iphone = new Product('Apple iPhone 7', 50000, 10);
3 $galaxy = new Product('Samsung Galaxy S7', 50000);
4
5 //и можно сразу работать со свойствами и методами
6 echo $iphone->price; //что выведет?
7 echo $iphone->name; //что выведет?
8 echo $galaxy->price; //что выведет?
9 echo $iphone->getPrice(); //что выведет?
```



---

# ПРО КОНСТРУКТОРЫ

1. В общем и целом, чаще всего конструкторы используются для предварительной инициализации объектов, чтобы с ними можно сразу было работать
2. При этом не стоит сразу в них пытаться передать все возможные параметры - старайтесь передавать то, что нужно для старта и без чего объект не имеет смысла
3. Общий смысл магических методов - реакция на какие-то события связанные с работой объектов. Конструктор вызывается при срабатывании события "создание объекта"

---

## ЧТО ЕЩЁ?

Всё круто, но есть ещё один момент. В нашем примере с продуктами для Интернет-магазина. Бизнес-логика такова, что мы обязаны везде выводить цену через метод `getPrice`, чтобы учитывать скидку.

Но никто не мешает обращаться к свойству `price` напрямую и, что ещё более печально, изменять его. Никто не застрахован от человеческого фактора.

А вот мы попробуем застраховаться

---

# ОБЛАСТЬ ВИДИМОСТИ СВОЙСТВ И МЕТОДОВ



# ОБЛАСТИ ВИДИМОСТИ

Из того, что мы уже использовали, остался один вопрос - что же такое public?

В рамках класса свойства и методы могут иметь свою область видимости, которая определяет, как работать с этими свойствами

1. private
2. protected
3. public

# PUBLIC

Public-свойства и методы, это как раз те свойства и методы, к которым мы можем обратиться в программе напрямую

```
1 class SimpleClass
2 {
3     public $publicProperty; // публичное свойство
4
5     //публичный метод
6     public function publicMethod() {}
7 }
```

## PUBLIC - КАК РАБОТАТЬ

```
1 $object = new SimpleClass();  
2 //вызываем публичный метод  
3 $object->publicMethod();  
4 //задаем публичное свойство  
5 $object->publicProperty = 100;  
6 //выводим значение публичного свойства  
7 echo $object->publicProperty;
```

И всё круто.

# PRIVATE И PROTECTED

Принципиальная разница между `private` и `protected` будет понятна на следующей лекции (наследование)

`Private` свойства принадлежат только классу, в котором были определены

`Protected` - принадлежат классу и наследникам

Не вдаваясь в детали, в обоих случаях к таким свойствам и методам **нельзя** обратиться напрямую.

# PRIVATE И PROTECTED

```
1  class SimpleClass
2  {
3      public $publicProperty; // публичное свойство
4      private $privateProperty; // приватное свойство
5      protected $protectedProperty; // защищенное свойство
6
7      public function publicMethod() {} //публичный метод
8      private function privateMethod() {} // приватный метод
9      protected function protectedMethod() {} //защищенный
10 }
```



## PRIVATE И PROTECTED (4.2)

```
1 $object = new SimpleClass(); //создали объект
2 $object->publicProperty = 100; // ок - изменили публичное
3 echo $object->publicProperty; // ок - вывели публичное се
4 $object->publicMethod(); // ок - обратились к публичному
5
6 $object->privateProperty = 100; // ошибка нельзя менять
7 echo $object->privateProperty; // ошибка даже получить е
8 $object->protectedProperty = 100; // ошибка нельзя менять
9 echo $object->protectedProperty; // ошибка вывести его т
10 $object->privateMethod(); // ошибка нельзя обратиться к
11 $object->protectedMethod(); // ошибка нельзя обратиться
```

# КАК РАБОТАТЬ С PRIVATE И PROTECTED

При этом данные свойства недоступны только для обращения извне. Внутри методов, например, мы всё также имеем возможность их изменять (через `$this`)

```
1 class SimpleClass
2 {
3     private $privateProperty = 100;
4
5     public publicMethod($value)
6     {
7         $this->privateProperty = $value; //всё ок
8     }
9 }
```

## КАК РАБОТАТЬ С PRIVATE И PROTECTED (Ч.2)

```
1  $object = new SimpleClass();  
2  //тут через публичный метод мы присвоим значение  
3  //приватному свойству  
4  $object->publicMethod(200);
```

Что? Почему? Потому что мы работаем в рамках одного контекста. А private и protected - защита от обращения вне его.

Поэтому легко можно определить публичные методы, которые будут отдавать приватные свойства.

# ПРИМЕР

```
1  class SimpleClass
2  {
3      private $privateProperty = 100;
4
5      //их ещё называют геттерами
6      public getPrivateProperty()
7      {
8          return $this->privateProperty;
9      }
10 }
```

## ЭМ. А РАВЗЕ ЭТО НЕ ДЫРКА В БЕЗОПАСНОСТИ?

Отчасти. Но:

1. Мы сами решаем для каких свойств предусматривать геттеры и сеттеры, а для каких нет. Что-то может остаться "внутренней кухней".
2. Мы можем предусмотреть дополнительную логику при обращении (например, логировать или считать сколько раз обратились к какому свойству)
3. Иными словами - мы имеем больше контроля, рамки которого определяем сами. В случае с публичными свойствами у нас контроля нет вообще. Любой в любом месте программы может их получать и менять.

## НА НАШЕМ ПРИМЕРЕ

В нашем случае логично использовать private для свойства цена. При этом мы должны иметь возможность разово её задать и всё.

```
1 class Product
2 {
3     private $price;
4     // тут свойства name, discount, category
5     public function __construct($name, $price, $discount
6     {
7         $this->name = $name;
8         $this->price = $price; // задали значение приватно
9         $this->discount = $discount;
10
```

## НА НАШЕМ ПРИМЕРЕ (Ч.2)

В нашем случае логично использовать `private` для свойства `цена`. При этом мы должны иметь возможность разово её задать и всё.

```
1 $iphone = new Product('Apple iPhone 7', 50000);
2 $iphone->price = 0; // ошибка - не надо так
3 echo $iphone->price; // ошибка - атата, так цену не получ
4 //но свойство price имеет значение (присвоено в конструкт
5 // поэтому мы легко можем вывести цену, как нам нужно
6 $iphone->getPrice(); //все ок и это единственный способ е
```

Да, сложно.



## НЕСКОЛЬКО СОВЕТОВ

1. Если ничего не понятно, задавайте вопросы.
2. Если всё равно не понятно, попробуйте вернуться к этому завтра
3. При создании своих классов для простоты сперва используйте `public`
4. По мере развития класса - вы поймёте, что следует закрыть
5. Используя области видимости вы ясно определяете правила, по которым класс работает. В результате, больше шансов, что человек, который будет работать с вашим кодом будет использовать его правильно.





## НО И ЭТО ЕЩЁ НЕ ВСЁ...

На самом деле, мы рассмотрели почти всё, что нужно для начала, но есть ещё несколько базовых моментов, о которых непременно нужно знать



# СТАТИЧЕСКИЕ МЕТОДЫ И СВОЙСТВА

# СТАТИЧЕСКИЕ СВОЙСТВА

Те методы и свойства, что мы уже рассмотрели, были свойствами **объектов**. Они существуют в контексте объекта и меняют его поведение.

Помните аналогию с формой для печенек?

Но ООП пошло дальше и предоставило ещё один инструмент - статические свойства и методы. Начнём со свойств.

## СТАТИЧЕСКИЕ СВОЙСТВА (Ч.2)

Статические свойства объявляются через ключевое слово `static` и являются свойствами класса, а не объекта

```
1 class SimpleClass
2 {
3     public static $staticProperty = 0; //статическое свойство
4     //которое также может иметь область видимости
5 }
6
7 echo SimpleClass::$staticProperty; //вот так можно
8 $object = new SimpleClass();
9 $object::$staticProperty; //а вот так нельзя
```

## СТАТИЧЕСКИЕ СВОЙСТВА (Ч.3)

К статическим свойствам можно обращаться внутри методов. Но через `self::`

```
1 class SimpleClass
2 {
3     public static $staticProperty = 0; //статическое свойство
4     public function publicMethod()
5     {
6         echo self::$staticProperty;
7     }
8 }
9 $object = new SimpleClass();
10 $object->publicMethod(); //выведет 0
```

# СТАТИЧЕСКИЕ МЕТОДЫ

Со статическими методами всё также, но к ним можно обращаться из объекта.

```
1 class SimpleClass
2 {
3     public static $staticProperty = 0; //статическое свойство
4     public static function staticMethod() //статический метод
5     {
6         echo self::$staticProperty;
7     }
8 }
9 $object = new SimpleClass();
10 $object::staticMethod(); //вот так тоже можно
```

---

## ЧТО ТАКОЕ SELF?

Помните, мы говорили, что, по сути, `$this` - это ссылка на конкретный объект?

`self` - это ссылка на сам класс, т.е. мы можем работать с теми или иными методами без необходимости вообще создавать объекты

В связи с этим есть ещё одна особенность - в статических методах **нет** возможности обратиться к `$this`, т.к. по своей природе статические свойства существуют и могут работать без создания экземпляров класса

## ЧТО ТАКОЕ SELF? (4.2)

```
1  class SimpleClass
2  {
3      public $publicProperty;
4      public static $staticProperty = 0; //статическое свойство
5      public static function staticMethod() //статический метод
6      {
7          echo self::$staticProperty; //всё ок
8          echo $this->publicProperty; // ошибка - нет доступа
9      }
10 }
```



# НЕБОЛЬШОЙ ЛАЙФХАК

А теперь совсем интересно - изменяя статическое свойство, мы изменяем его для всех **объектов**. Круто, да?

```
1 class SimpleClass
2 {
3     public static $staticProperty = 0; //статическое свойство
4
5     public function __construct()
6     {
7         //при вызове конструктора меняем статическое свойство
8         self::$staticProperty++;
9     }
10 }
```

## НЕБОЛЬШОЙ ЛАЙФХАК (Ч.2)

```
1  echo SimpleClass::$staticProperty; //выведет 0, всё ок
2
3  $object = new SimpleClass(); //тут вызвался конструктор
4  $anotherObject = new SimpleClass(); //и тут вызвался конс
5  $thirdObject = new SimpleClass(); // и даже тут
6
7  echo SimpleClass::$staticProperty; // выведет 3
```



## ПЫТАЕМСЯ ПОНЯТЬ

1. Статические свойства и методы принадлежат классу, а не объекту. Объект может их менять, но меняет в классе. Как следствие, это свойства, которые для всех объектов общие.
2. Скорее всего вам не понадобятся статические свойства и методы в ближайшем будущем, но помнить об этом нужно
3. В ряде случаев есть необходимость добавить метод, который никак не завязан на внутренние параметры. Статические методы - хороший способ.
4. В ряде случаев вам захочется собрать логику в одном месте, но создание объекта будет избыточным (например, запрос)



## ПЫТАЕМСЯ ПОНЯТЬ (Ч.2)

1. Как это использовать на практике? Ну, на нашем примере, мы можем захотеть давать скидку на каждый второй товар. И для этого нужно понимать, сколько вообще товаров было добавлено в корзину.
2. А вообще - использование статических свойств нужно крайне редко. Я бы рекомендовал так: если в методе не используются внутренние свойства объекта - делайте его статическим. Но даже это не обязательно.



# КОНСТАНТЫ КЛАССОВ

# КОНСТАНТЫ КЛАССОВ

Константы классов - по сути, это статические свойства, которые нельзя изменять.

```
1 class SimpleClass
2 {
3     const CONSTANT = 'Значение константы';
4
5     public function showConstant() {
6         echo self::CONSTANT;
7     }
8 }
9 echo SimpleClass::CONSTANT; // всё ок
```

## КОНСТАНТЫ КЛАССОВ (Ч.2)

Как и с обычными константами, если их пытаться изменять - будет ошибка

```
1 echo SimpleClass::CONSTANT; // всё ок
2 SimpleClass::CONSTANT = 100; // ошибка
```

Зачем делать константы в классе, а не в общей программе? Если они относятся к общему контексту, то делать их в классе - правильнее. Да, да. Инкапсуляция.



**ЧТО ЕЩЁ НУЖНО ЗНАТЬ?**



# НЕКОТОРЫЕ НЮАНСЫ

Объекты всегда передаются по ссылке.

```
1  class SimpleClass
2  {
3      public $publicProperty = 0;
4  }
5
6  //отдельно функция
7  function increment($object)
8  {
9      $object->publicProperty++;
10 }
```

## НЕКОТОРЫЕ НЮАНСЫ (Ч.2)

```
1 $object1 = new SimpleClass();
2 $object2 = new SimpleClass();
3
4 echo $object2->publicProperty; // 0, значение по умолчанию
5 increment($object2);
6 echo $object2->publicProperty; // вот тут будет 1, а не 0
```



## НЕКОТОРЫЕ НЮАНСЫ (Ч.3)

Очевидно, но не хватало таких примеров. Методы можно вызывать в других методах.

## НЕКОТОРЫЕ НЮАНСЫ (Ч.4)

```
1  class SimpleClass
2  {
3      public function publicMethod()
4      {
5          echo 'Hello ';
6          $this->anotherMethod();
7      }
8
9      public function anotherMethod()
10     {
11         echo 'world';
12     }
```



## НЕКОТОРЫЕ НЮАНСЫ (4.5)

Если вам еще не рассказывали, существует стандарт PSR.

1. Называйте класс всегда с большой буквы, и каждое слово в нём  
НапримерВотТак
2. Методы должны начинаться с маленькой буквы напримерВотТак
3. Фигурные скобки - каждая на новой строке для методов и классов



Спасибо за внимание!

# ЕВГЕНИЙ КОРЫТОВ



[korytoff@gmail.com](mailto:korytoff@gmail.com)



[korytoff](https://www.telegram.me/korytoff)