

НАСЛЕДОВАНИЕ И ИНТЕРФЕЙСЫ



ЕВГЕНИЙ КОРЫТОВ / РУКОВОДИТЕЛЬ ГРУППЫ ВЕБ-РАЗРАБОТКИ, CONDÉ NAST



ЕВГЕНИЙ КОРЫТОВ

Руководитель группы веб-разработки,
Condé Nast



korytoff@gmail.com



korytoff



ЧТО МЫ ЗНАЕМ?

- Классы
- Объекты
- Методы и свойства
- Три принципа ООП
- Область видимости методов и свойств
- Статические методы и свойства



НАСЛЕДОВАНИЕ



ТРИ ПРИНЦИПА ООП

- Инкапсуляция
- Полиморфизм
- Наследование

СРАЗУ К КОДУ (НАСЛЕДОВАНИЕ)

Простой пример наследования

```
1 class ParentClass //суперкласс
2 {
3     // любые свойства и методы
4 }
5 class ChildClass extends ParentClass
6 {
7     // данный класс наследует все свойства и методы ParentClass
8     //+ может определять свои свойства и методы
9 }
```

ПРО УТОК

Утка - птица, которая крякает. Пингвин - птица, которая не летает...

Птица - является суперклассом.

ПРО УТОК (КОД)

```
1  class Bird //суперкласс
2  {
3      public $color = 'black'; //предположим, что большинст
4      public function fly() { echo 'Я летаю'; }
5  }
6  class Duck extends Bird
7  {
8      //наследует все свойства и методы Bird
9  }
10
11  $duck = new Duck();
12  $duck->fly(); // выдаст 'Я летаю'
```




ПРО АЛЬБИНОСОВ, МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ И СУПЕРКЛАССЫ

Плохая новость: в PHP нет множественного наследования. Наследовать свойства и методы можно только от одного класса.

Хорошая новость: НО. Цепочка наследуемых классов может быть сколь угодно длинной.

Далее рассмотрим на альбиносах

ПРО АЛЬБИНОСОВ (КОД) - Ч.1

```
1  class Bird // суперкласс
2  {
3      public $sound; //пока никакого звука
4      public function fly()
5      {
6          echo 'Я летаю';
7      }
8      public function makeSomeSound()
9      {
10         echo 'Я издаю звук ' . $this->sound;
11     }
12 }
```

ПРО АЛЬБИНОСОВ (КОД) - Ч.2

```
1  class Duck extends Bird
2  {
3      public $sound = "Кря"; // + наследует все свойства и
4  }
5  class AlbinoDuck extends Duck
6  {
7      //наследует все свойства Duck + Bird
8  }
9  $duck = new Duck();
10 $duck->makeSomeSound(); // обычная утка говорит "Кря"
11 $albinoDuck = new AlbinoDuck();
12 $albinoDuck->makeSomeSound(); // Утка-альбинос тоже кряка
```



КАК - ПОНЯТНО. А НАФИГА?

- Мы можем выделить общие свойства у наших объектов, вынести их в отдельный тип.
- Если у объектов есть общее поведение, мы избегаем дублирования кода. Делаем его более читаемым и понятным.
- В самом простом случае мы можем описать общие моменты в суперклассе, а в остальных классах описывать только специфическую логику.

ПЕРЕГРУЗКА МЕТОДОВ

Важным моментом является возможность создать уникальное поведение. Предположим, наш альбинос отличается не только цветом, но и издаёт иные звуки. Мы можем **перегрузить** свойства и методы родителя.

ПЕРЕГРУЗКА МЕТОДОВ (КОД) - Ч.1

```
1 class Bird // суперкласс
2 {
3     public $sound; //пока никакого звука
4     public $color; //добавили новое свойство - цвет
5     public function fly() { echo 'Я летаю'; }
6     public function makeSomeSound()
7     {
8         echo 'Я издаю звук ' . $this->sound;
9     }
10 }
```

ПЕРЕГРУЗКА МЕТОДОВ (КОД) - Ч.2

```
1 //класс перегружает свойство $sound
2 class Duck extends Bird { public $sound = "Кря" }
3
4 class AlbinoDuck extends Duck //наследует Duck + Bird
5 {
6     public $color = "white"; //перегружаем свойство
7     public function makeSomeSound() //перегружаем метод
8     {
9         echo "Нет времени объяснять! Я говорю мяу!";
10    }
11 }
```

ПЕРЕГРУЗКА МЕТОДОВ (КОД) - Ч.3

```
1  $duck = new Duck();
2  $duck->makeSomeSound(); // обычная утка говорит “Кря”
3  $albinoDuck = new AlbinoDuck();
4  // Утка-альбинос не только отличается цветом, но и говори
5  $albinoDuck->makeSomeSound(); //... и ей нет времени объяс
```


БЛА-БЛА-БЛА, А ЕСЛИ НЕ НА УТКАХ?

Задача: Предположим, мы хотим работать с товарами интернет-магазина через объекты. В самом простом виде, для начала, мы хотим, чтобы у товара было несколько свойств - название, цена. Для этого создаём простой класс:

Попробуем решить задачу двумя способами - без наследования и с наследованием.

Изначально решаем без наследования.

А ЕСЛИ НЕ НА УТКАХ? (КОД)

```
1  class Product
2  {
3      public $price; //цена
4      public $title; //название
5
6      public function __construct($title, $price)
7      {
8          $this->title = $title;
9          $this->price = $price;
10     }
11 }
```

УСЛОЖНЯЕМ ЗАДАЧУ

Задача (продолжение): Мы создали общую структуру. Теперь предположим, что наш магазин торгует книгами и важным свойствами являются количество страниц и автор книги. Предположим, что в дальнейшем, мы захотим вывести их в описании

Мы пока не знаем, что такое наследование. Нам нужно расширить имеющийся класс. Заодно, можем его переименовать.

УСЛОЖНЯЕМ ЗАДАЧУ (КОД)

```
1  class Book
2  {
3      public $price;
4      public $title;
5      public $pages;
6      public $author;
7
8      public function __construct($title, $price, $author,
9      {
10         //присваиваем соответствующие свойства
11     }
12 }
```

НО И ЭТО НЕ ВСЁ

Задача (продолжение): Всё вроде пока неплохо, верно? И зачем только придумали это дурацкое наследование. Но теперь владелец магазина решил, что книги нынче не в моде. И нужно расширять ассортимент. Начать решил с чего-то простого. Например, с флэшек (почему бы и нет?). У них вообще нет параметров страниц. Зато важным является объём памяти.

Что же, добавляем новый класс для флэш-накопителей. (а куда деваться?)

НО И ЭТО НЕ ВСЁ (КОД)

```
1  class Flash
2  {
3      public $price;
4      public $title;
5      public $memory;
6
7      public function __construct($title, $price, $memory)
8      {
9          $this->title = $title;
10         $this->price = $price;
11         $this->memory = $memory
12
```



НЕ ТОМИ! ГДЕ НАСЛЕДОВАНИЕ?

Задача (продолжение): Но и тут нет веских причин что-то наследовать. Но тут выясняется, что мы забыли одну деталь. В обоих случаях нам нужно отдельно вывести цену и отдельно вывести название товара. Т.е. добавить пару методов в пару классов

НЕ ТОМИ! ГДЕ НАСЛЕДОВАНИЕ? (КОД)

```
1  class Flash
2  {
3      //тут все методы и свойства с предыдущего примера
4      public function getTitle()
5      {
6          return $this->title;
7      }
8      public function getPrice()
9      {
10         return $this->price;
11     }
12 }
```


А ПОЧЕМУ БЫ И НЕ ДА?

Аналогичный код мы добавим в класс Book. А теперь давайте порассуждаем.

- Что нужно сделать, если мы решим торговать ещё каким-то товаром?
- Что если мы захотим изменить метод title?
- Есть ли дублирование кода? Удобно ли это?
- Как нам может помочь наследование?

НАСЛЕДОВАНИЕ: ВОЗВРАЩЕНИЕ

Хорошо, попробуем улучшить наш код. Возвращаемся к первому варианту и немного расширим его. Для начала нам нужен общий суперкласс `Product`, который будет содержать свойства и методы, общие для всех товаров.

Обращаю внимание - общие для всех товаров. Т.е. если в будущем нам понадобятся иные расширения для наших частных товаров, мы будем расширять именно этот класс, а дочерние классы будут автоматически подхватывать эти свойства.

НАСЛЕДОВАНИЕ (КОД)

```
1  class Product
2  {
3      public $price; // цена есть у любого товара
4      public $title; // название - тоже
5      public function __construct($title, $price) { //присе
6      public function getPrice()
7      {
8          return $this->price;
9      }
10     public function getTitle() { return $this->title; }
11 }
```

ВОПРОСЫ К РОДИТЕЛЮ

Ни у кого не возникло вопросов к родителю? Раньше, мы через конструктор передавали все соответствующие параметры товара, а теперь только цену и название.

На самом деле это даже хорошо и правильно. Мы не будем перегружать конструктор, а воспользуемся геттерами и сеттерами.

А заодно получим порцию небольшого синтаксического сахара :)

ВОПРОСЫ К РОДИТЕЛЮ

```
1  class Book extends Product //наследуем
2  {
3      //все свойства и методы Product уже есть
4      //мы описываем только уникальные особенности
5      public $author;
6      public $pages;
7      public setYear($year)
8      {
9          $this->year = $year;
10         return $this; //магия, о которой мы поговорим чуть
11     }
12
```



RETURN \$THIS ???

Знакомимся: fluent setters. Это то, что позволяет нам писать код более красиво. Вместо того, чтобы определять параметры в конструкторе, мы можем задавать свойства книги чуть иначе

ОБЫЧНЫЕ СЕТТЕРЫ

```
1 //предположим мы определили сеттеры для каждого свойства
2 $book = new Book('Мастер и Маргарита', 500);
3 $book->setAuthor('Михаил Булгаков');
4 $book->setYear(2015); //дата выпуска
5 $book->setPages(200);
6 //любые другие свойства мы будем задавать отдельно в кажд
```

FLUENT SETTERS

```
1 //предположим мы определили сеттеры для каждого свойства
2 $book = new Book('Мастер и Маргарита', 500);
3 $book->setAuthor('Михаил Булгаков')->setYear(2015)->setPa
4
5 //или (идентично)
6 $book->setAuthor('Михаил Булгаков')
7     ->setYear(2015)
8     ->setPages(200);
```

Этим можно пользоваться не только для сеттеров.

ВЕРНЕМСЯ К КНИГАМ

Ввиду того, что кода вмещается мало на один слайд, перейдем к реалтайм кодингу.

Попробуем решить нашу задачу в реальном времени и получить полную картину. Заодно дополним нашу задачу. Для расчета стоимости доставки нам понадобился вес. Попробуем решить, куда мы его будем добавлять и добавим метод для расчета доставки.

Если вы вдруг читаете этот слайд и хотите посмотреть, что было дальше - добро пожаловать в видео %)



АБСТРАКТНЫЕ КЛАССЫ

А ДАВАЙТЕ ДОБАВИМ ОПИСАНИЯ?

Что же. Теперь перед нами стоит новая задача. Нужно, чтобы каждый тип продукта имел метод `getDescription`, который бы возвращал описание, уникальное для каждого типа продукта.

Например, в случае с флэшками, это может быть: "Флэшка на N мегабайт", где N, мы возьмем из соответствующего свойства.

Но для начала, давайте разберемся, что такое абстрактные классы и методы. И как они могут нам пригодиться.

АБСТРАКТНЫЕ КЛАССЫ И МЕТОДЫ

Говоря по простому, абстрактные методы - это методы, не имеющие реализации.

```
1 // класс, содержащий абстрактные методы, является абстрак
2 abstract class AbstractClass
3 {
4     abstract function AbstractMethod(); //абстрактный ме
5     //...у которого нет тела
6 }
```



АБСТРАКТНЫЕ КЛАССЫ И МЕТОДЫ

Некоторые свойства абстрактных классов

- Мы не можем создать объект абстрактного класса (т.к. есть методы, которые не имеют реализации)
- Абстрактный класс может содержать, как абстрактные методы, так и конкретные реализации.

На базе абстрактных методов строятся и паттерны проектирования, но об этом - в другой раз



О_О ЗАЧЕМ?

По правде говоря, это лишь возможность организовать свой код так, как задумано.

Например, для примера выше - если мы добавим в класс Product конкретное описание по умолчанию, которое унаследуют все классы, мы можем забыть описать его в одном из дочерних классов и получить не ожидаемое поведение.

ВЕРНЕМСЯ К ПРИМЕРУ

```
1  class Product
2  {
3      //тут все остальные свойства и методы
4
5      public function getDescription()
6      {
7          // реализация по умолчанию, которую получают все д
8          // но наша задача - обязать переопределять это по
9          return '';
10     }
11 }
```

ВЕРНЕМСЯ К ПРИМЕРУ (С АБСТРАКТНЫМ КЛАССОМ)

```
1  abstract class Product //класс объявлен абстрактным
2  {
3      public $price; // цена есть у любого товара
4      public $title; // название - тоже
5
6      // тут все остальные методы
7
8      abstract public function getDescription(); //абстракт
9  }
```

В ЧЕМ ПРОФИТ?

Теперь мы точно не забудем определить в Книгах и Флэшках описание - если не определить в них конкретную реализацию для абстрактных методов, `php` тихохонько ругнётся при попытке создания объекта. Или не тихохонько. Но работать не будет.

Вообще всю суть и пользу абстрактных классов нужно прочувствовать. Чаще всего они вам не понадобятся, но когда вы столкнетесь с соответствующей задачей, будете радоваться, что они существуют :)

В ЧЕМ ПРОФИТ? (КОД)

```
1  class Flash extends Product
2  {
3      public $memory;
4      //тут остальные свойства и методы
5
6      //описываем реализацию абстрактного метода
7      public function getDescription()
8      {
9          return "Флэшка на {$this->memory} мегабайт"
10     }
11 }
```



НУ МАААААМ! ИЛИ КЛЮЧЕВОЕ СЛОВО PARENT

Хорошо, а теперь, предположим наша задача поменялась. Мы провели маркетинговое исследование и поняли, что в описание нужно обязательно включать цену. Например, большинство клиентов звонят в call-центр и уточняют её, т.к. цена на сайте расположена неудобно.

Сначала, забудем на секунду об абстрактных классах и попробуем решить проблему максимально просто.

Для этого переделаем наш класс Product.

НУ МАААААМ! ИЛИ КЛЮЧЕВОЕ СЛОВО PARENT (КОД)

```
1 class Product //класс перестал быть абстрактным
2 {
3     //тут остальные свойства и методы
4
5     public function getDescription() //это уже НЕ абстракт
6     {
7         echo "Цена: {$this->price}"; // по умолчанию в ог
8     }
9 }
```

КАК СДЕЛАТЬ ТАК, ЧТОБЫ ВСЕМ БЫЛО ХОРОШО?

Проблема следующая - если мы просто перегрузим метод в дочерних классах, вывод цены мы тоже перетрем. И нам придется дублировать код. А если мы захотим там же выводить и название в описании для всех товаров в будущем? Изменения потребуются в трех местах (двух дочерних и одном родительском классе).

Нам это не подходит. НО. Мы можем обращаться к реализации любого метода родителя через ключевое слово `parent`

НУ МАААААМ! ИЛИ КЛЮЧЕВОЕ СЛОВО PARENT (КОД)

```
1 class Flash extends Product
2 {
3     //тут остальные свойства и методы
4
5     public function getDescription()
6     {
7         parent::getDescription(); // обратились к методу
8         //добавили к описанию родителя всё, что хотим
9         echo "Флэшка на {$this->memory} мегабайт"
10
```

НО... КАК ЖЕ АБСТРАКЦИЯ?

Выше мы говорили, что абстракция позволяет избежать нам ошибок. По крайней мере мы требуем добавления уникальных свойств каждого продукта в описание.

Конечно, ничего критичного (описание всегда будет минимум содержать цену), но мы ожидаем, что описание будет уникально для каждого типа продукта.

Проведем небольшой рефакторинг. Он поможет лучше понять, как еще можно работать с абстрактными классами и каким вторым способом можно было решить задачу.

НО... КАК ЖЕ АБСТРАКЦИЯ? (КОД)

```
1  abstract class Product //и снова класс объявлен абстрактн
2  {
3      //тут остальные свойства и методы
4      public function getFullDescription()
5      {
6          echo "Цена: {$this->price}"; // по умолчанию в ог
7          $this->getDescription(); // да, да мы обращаемся
8      }
9      abstract public function getDescription(); //абстракт
10 }
```




ЧТО У НАС ПОЛУЧИЛОСЬ?

Это позволяет нам не обращаться в дочерних классах к родителю. Достаточно реализовать метод `getDescription`, а запрашивать описание через `getFullDescription`.

При этом мы закрыли обе проблемы - мы не забудем добавить уникальное описание, и наше общее описание всегда будет содержать всю нужную информацию.



ЧТО ДЕЛАТЬ, ЕСЛИ ЧТО-ТО НЕПОНЯТНО?

- Задавать вопросы
- Пытаться осознать и прочувствовать
- После лекции не лениться практиковаться

Но сейчас удобнее всего задавать вопросы)



КЛЮЧЕВОЕ СЛОВО FINAL

А ТЕМ ВРЕМЕНЕМ, МЫ ПРОДОЛЖАЕМ

Согласитесь, лучше бы про уток?) Ну да ладно. Есть ещё один момент от которого хочется закрыться. Например, нам не нужно, чтобы от класса Book наследовались какие-то свойства. Чтобы ненароком, другой программист не вздумал наследовать флэшки от книг (в жизни бывает всякое).

В общем виде - архитектурно - длинные цепочки наследования увеличивают сложность системы. Изменения в родителе влияют на всех потомков. Поэтому при сложной структуре можно получить ошибки. Это не значит, что это плохо. Но нужно правильно их использовать.

А ТЕМ ВРЕМЕНЕМ, МЫ ПРОДОЛЖАЕМ

Согласитесь, лучше бы про уток?) Ну да ладно. Есть ещё один момент от которого хочется закрыться. Например, нам не нужно, чтобы от класса Book наследовались какие-то свойства. Чтобы ненароком, другой программист не вздумал наследовать флэшки от книг (в жизни бывает всякое).

В общем виде - архитектурно - длинные цепочки наследования увеличивают сложность системы. Изменения в родителе влияют на всех потомков. Поэтому при сложной структуре можно получить ошибки. Это не значит, что это плохо. Но нужно правильно их использовать.



А ТЕМ ВРЕМЕНЕМ, МЫ ПРОДОЛЖАЕМ

В связи с вышесказанным - поэтому последний уровень нашей цепочки всегда стоит закрывать, если мы не хотим его расширения

Это делается при помощи ключевого слово `final`

FINALITY

```
1 //теперь никто не сможет наследовать методы и свойства да  
2 final class Flash extends Product  
3 {  
4     //тут остальные свойства и методы  
5 }
```

При этом окончательным может быть не только класс, но и конкретный метод. Это запрещает его перегрузку.

И ЕЩЕ ПРО ОБЛАСТИ ВИДИМОСТИ



СО ЩИТОМ ИЛИ НА ЩИТЕ

Предполагается, что с прошлой лекции вы помните, чем отличаются данные области видимости свойств или методов. В данном случае, мы обзорно вспомним и акцентируем внимание на `private` и `protected` областях видимости

PUBLIC

Можно получить доступ к свойству или методу отовсюду (внутри класса, внутри родительского класса. В общем виде, мы можем изменять значения `memory` для наших флэш-карт и запрашивать их и без геттеров и сеттеров

```
1 $flash = new Flash('Флэшка', 100);  
2 $flash->memory = 2000;  
3 echo $flash->memory; // выведет 2000
```

Это не совсем корректно, т.к. можно случайно изменить свойство класса, а в нашем случае оно должно быть неизменяемым.

PRIVATE

Доступ к таким методам и свойствам возможен только внутри самого класса. Классы, которые его наследуют, не получают доступ к данному методу или свойству.

В случае с `memory` это как раз то, что нам нужно. Мы не планируем, что у класса вообще будут наследники, а если мы и забыли ключевое слово `final`, мы не дадим другим классам работать с этим свойством.

PRIVATE (КОД)

```
1  final class Flash extends Product
2  {
3
4      private $memory; // в данном случае мы запрещаем кому
5      // к данному свойству извне
6
7      //при этом, если мы хотим изменить свойство - есть се
8      public function setMemory($memory)
9      {
10         $this->memory = $memory;
11     }
12 }
```

PRIVATE (КОД)

```
1 $flash = new Flash( 'Флэшка', 100 );  
2 $flash->memory = 2000; // выдаст ошибку, пытаемся изменить  
3 $flash->setMemory = 2000; // а так можно (если мы оставял
```

Изменение значения свойства через сеттер все равно имеет ряд преимуществ. Например, мы можем не давать присваивать отрицательные значения. В случае работы с public свойством напрямую, мы это ограничить не можем.

PROTECTED

Но что делать с ценой и названием товара? Если вы помните, у нас они также объявлены, как публичные свойства. Соответственно в любом месте кода можно обратиться к свойству напрямую и изменить его. Это может обеспечить проблемы в будущем.

При этом мы не можем объявить их как приватные переменные. В этом случае, наши конкретные товары не будут иметь доступ к этим свойствам.

Есть два варианта решения проблемы: Объявить эти свойства приватными, но написать для них геттеры и сеттеры или Объявить эти свойства защищенными (`protected`)

PROTECTED

Второй способ для них наиболее предпочтителен, т.к. мы можем в будущем захотеть оперировать ценой в каждом классе по своему. Protected не позволит обращаться к цене напрямую

```
1 $flash = new Flash( 'Флэшка', 100 );  
2 $flash->price = 200; // выдаст ошибку, пытаемся изменить
```

Но! При этом мы сможем работать внутри класса flash со свойством, как захотим. Например, мы хотим, чтобы на все флэшки действовала скидка 20 процентов.

PROTECTED (КОД)

```
1  final class Flash extends Product
2  {
3      // остальные свойства и методы
4
5      //перегружаем геттер для получения цены конкретно для
6      public function getPrice()
7      {
8          //саму скидку по хорошему тоже вынести в свойство
9          //но в какой класс?
10         return round($this->price * 20 / 100)
11     }
12 }
```




ИНТЕРФЕЙСЫ

ИНТЕРФЕЙСЫ

Еще раз вспомним, что есть абстрактные классы. Это классы, у которых могут быть методы вообще без реализации.

Интерфейсы объявляются, как и классы, но вместо ключевого слова класс используется слово `interface`

```
1  interface SimpleInterface
2  {
3      // мы объявили пустой интерфейс
4  }
```

ИНТЕРФЕЙСЫ

Интерфейс может содержать **только методы**, при чем без их реализации и **константы**. При этом описывается сигнатура методов (какие параметры они должны принимать)

```
1 interface SimpleInterface
2 {
3     //интерфейс не содержит свойств
4     //методы не содержат реализации
5     public function someFunction($argument1, $argument2);
6     public function secondFunction($argument1);
7     public function anotherFunction();
8 }
```

ИНТЕРФЕЙСЫ

Класс может реализовывать интерфейс (impleментировать). Если класс делает это он ОБЯЗАН определить реализацию конкретных методов интерфейса в соответствии с их сигнатурой

```
1 interface SimpleInterface
2 {
3     public function someFunction($argument1, $argument2);
4 }
5 class SimpleClass implements SimpleInterface
6 {
7     //описываем, что должен делать этот метод, в данном с
8     public function someFunction($argument1, $argument2)
9 }
```



ЕЩЕ НЕМНОГО ПРО ИНТЕРФЕЙСЫ

- Интерфейс может наследовать другой интерфейс (extends)
- Все методы интерфейсов должны быть публичными
- Класс не может имплементировать два интерфейса, содержащих один и тот же метод
- Класс может имплементировать какое угодно количество интерфейсов



СТОП... СТОООООП! ОМГ, А ЭТО ЗАЧЕМ?!

- Вы можете писать свой код и без интерфейсов. Интерфейсы помогают лучше его структурировать, добавляя обязательные элементы для реализации. Меньше шансов сделать ошибку.
- Интерфейсы могут являться типами аргументов

ОМГ, А ЭТО ЗАЧЕМ?! (КОД)

```
1 interface SimpleInterface
2 {
3     public function someFunction($argument1, $argument2);
4 }
5 class SimpleClass implements SimpleInterface
6 {
7     //описываем, что должен делать этот метод, в данном с
8     public function someFunction($argument1, $argument2)
9 }
```

ОМГ, А ЭТО ЗАЧЕМ?! (КОД) - Ч.2

```
1  class AnotherSimpleClass
2  {
3      // использование интерфейса гарантирует, что в object
4      // someFunction, иначе возникнет фатальная ошибка
5      public function saySmtth(SimpleInterface $object)
6      {
7          // мы уверен, что у object есть метод SomeFunction
8          // и спокойно его используем внутри этого метода
9          $object->someFunction(1, 2);
10     }
11 }
```

**КАК ПРИГОТОВИТЬ
ВКУСНОЕ БЛЮДО.
ПРИМЕСИ (TRAITS)**



ПРИМЕСИ (TRAITS)

Примеси (traits) - еще один способ расширить функциональность нашего класса не прибегая к наследованию.

Возвращаясь к нашему примеру, предположим, что мы хотим, чтобы для части товаров была возможность менять цену в процессе работы приложения, а у части - нет. И товаров у нас теперь не два, а десять. Магазин теперь торгует помимо книг

Можем ли мы использовать при этом наследование? Что произойдет если мы будем описывать геттеры и сеттеры в родительском классе?

ПРИМЕСИ (TRAITS)

Примеси позволяют вынести часть функционала и использовать его в любом классе

```
1  trait ChangePrice //обратите внимание, примесь это не класс
2  {
3      public function setPrice($price) { $this->price = $price; }
4  }
5  class SomeProduct extends Product
6  {
7      use ChangePrice; // подключаются примеси при помощи use
8      // данный класс получит все свойства и методы родителя
9      // плюс будет расширен методами и свойствами примеси
10 }
```

В ЧЕМ ФИШКА?

- Можно вынести дублирующуюся часть кода в отдельную примесь и подключать её только туда, куда нужно.
- Более удобный и читаемый код

```
1  class SomeClass
2  {
3      use ChangePrice, ChangeMemory, ChangeTitle; // эти пр
4      // данный класс “собран” из примесей и мы сразу понимаем
5      // что он может работать с ценой, памятью и названием
6  }
```



НА ПРАКТИКЕ

Принципиальной необходимости в примесях нет. Это просто еще один способ сделать ваш код удобней и читабельнее. Хотя злоупотребление ими тоже имеет последствия - можно запутаться в пересекающихся свойствах и методах.

Это способ организовать множественное наследование

На практике острой необходимости в них почти не возникает, но помните, что такая возможность существует.



Спасибо за внимание!

ЕВГЕНИЙ КОРЫТОВ



korytoff@gmail.com



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