ООП в PHP. Базовые понятия.

Оглавление

[Откуда появилась концепция ООП? 1](#_Toc462406462)

[Базовые понятия ООП 1](#_Toc462406463)

[Основные принципы ООП 2](#_Toc462406464)

[ООП в PHP 2](#_Toc462406465)

[Константы и статические методы 5](#_Toc462406466)

[Домашнее задание 6](#_Toc462406467)

[Дополнительные материалы 8](#_Toc462406468)

# 

Откуда появилась концепция ООП?

Увеличение размеров программ приводило к необходимости привлечения большего числа программистов, что, в свою очередь, потребовало дополнительных ресурсов для организации их согласованной работы. В процессе разработки приложений заказчик зачастую изменял функциональные требования, что еще более усложняло процесс создания программного обеспечения.

Но не менее важными оказались качественные изменения, связанные со смещением акцента использования компьютеров. В эпоху "больших машин" основными потребителями программного обеспечения были такие крупные заказчики, как большие производственные предприятия, финансовые компании, государственные учреждения. Стоимость таких вычислительных устройств для небольших предприятий и организаций была слишком высока.

Позже появились персональные компьютеры, которые имели гораздо меньшую стоимость и были значительно компактнее. Это позволило широко использовать их в малом и среднем бизнесе. Основными задачами в этой области являются обработка данных и манипулирование ими, поэтому вычислительные и расчетно-алгоритмические задачи с появлением персональных компьютеров отошли на второй план. Как показала практика, традиционные методы процедурного программирования не способны справиться ни с нарастающей сложностью программ и их разработки, ни с необходимостью повышения их надежности. Во второй половине 80-х годов возникла настоятельная потребность в новой методологии программирования, которая была бы способна решить весь этот комплекс проблем. Ею стало объектно-ориентированное программирование (ООП).

Объектно-ориентированный подход обладает такими преимуществами, как:

* уменьшение сложности программного обеспечения;
* повышение надежности программного обеспечения;
* обеспечение возможности модификации отдельных компонентов программного обеспечения без изменения остальных его компонентов;
* обеспечение возможности повторного использования отдельных компонентов программного обеспечения.

Базовые понятия ООП

Основной структурной единицей в ООП является объект.

**Объект** - это мыслимая или реальная сущность, обладающая характерным поведением и отличительными характеристиками и являющаяся важной в предметной области. Каждый объект имеет состояние, обладает четко определенным поведением и уникальной идентичностью.

**Состояние** (state) - совокупный результат поведения объекта: одно из стабильных условий, в которых объект может существовать, охарактеризованных количественно; в любой момент времени состояние объекта включает в себя перечень (обычно статический) свойств объекта и текущие значения (обычно динамические) этих свойств.

**Поведение** (behavior) - действия и реакции объекта, выраженные в терминах передачи сообщений и изменения состояния; видимая извне и воспроизводимая активность объекта.

**Identity** (уникальность) объекта состоит в том, что всегда можно определить, указывают две ссылки на один и тот же объект или на разные объекты. При этом два объекта могут во всем быть похожими, их образ в памяти может представляться одинаковыми последовательностями байтов, но, тем не менее, их Identity может быть различна.

Основные принципы ООП

ООП базируется на трёх основных принципах – наследование, инкапсуляция и полиморфизм.

**Наследование** (inheritance) - это отношение между классами, при котором класс использует структуру или поведение другого класса (одиночное наследование), или других (множественное наследование) классов. Наследование вводит иерархию "общее/частное", в которой подкласс наследует от одного или нескольких более общих суперклассов. Подклассы обычно дополняют или переопределяют унаследованную структуру и поведение.

Использование наследования способствует уменьшению количества кода, созданного для описания схожих сущностей, а также способствует написанию более эффективного и гибкого кода.

**Инкапсуляция** (encapsulation) - это сокрытие реализации класса и отделение его внутреннего представления от внешнего (интерфейса). При использовании объектно-ориентированного подхода не принято применять прямой доступ к свойствам какого-либо класса из методов других классов. Для доступа к свойствам класса принято задействовать специальные методы этого класса для получения и изменения его свойств.

Внутри объекта данные и методы могут обладать различной степенью открытости (или доступности).

Открытые члены класса составляют внешний интерфейс объекта. Это та функциональность, которая доступна другим классам. Закрытыми обычно объявляются все свойства класса, а также вспомогательные методы, которые являются деталями реализации и от которых не должны зависеть другие части системы.

Благодаря сокрытию реализации за внешним интерфейсом класса можно менять внутреннюю логику отдельного класса, не меняя код остальных компонентов системы. Это свойство называется модульность.

Обеспечение доступа к свойствам класса только через его методы также дает ряд преимуществ. Во-первых, так гораздо проще контролировать корректные значения полей, ведь прямое обращение к свойствам отслеживать невозможно, а значит, им могут присвоить некорректные значения.

Во-вторых, не составит труда изменить способ хранения данных. Если информация станет храниться не в памяти, а в долговременном хранилище, таком как файловая система или база данных, потребуется изменить лишь ряд методов одного класса, а не вводить эту функциональность во все части системы.

Наконец, программный код, написанный с использованием данного принципа, легче отлаживать. Для того, чтобы узнать, кто и когда изменил свойство интересующего нас объекта, достаточно добавить вывод отладочной информации в тот метод объекта, посредством которого осуществляется доступ к свойству этого объекта. При использовании прямого доступа к свойствам объектов программисту

пришлось бы добавлять вывод отладочной информации во все участки кода, где используется интересующий нас объект.

**Полиморфизм** (polymorphism) - положение теории типов, согласно которому имена (например, переменных) могут обозначать объекты разных (но имеющих общего родителя) классов. Следовательно, любой объект, обозначаемый полиморфным именем, может по-своему реагировать на некий общий набор операций.

ООП в PHP

Описывая поведение какого-либо объекта, например, автомобиля, мы строим его модель. Модель, как правило, не может описать объект полностью, реальные объекты слишком сложны. Приходится отбирать только те характеристики объекта, которые важны для решения поставленной перед нами задачи. Для описания грузоперевозок важной характеристикой будет грузоподъемность автомобиля, а для описания автомобильных гонок она не существенна. Но для моделирования гонок обязательно надо описать метод набора скорости данным автомобилем, а для грузоперевозок это не столь важно.

Мы должны *абстрагироваться* от некоторых конкретных деталей объекта. Очень важно выбрать правильную степень абстракции. Слишком высокая степень даст только приблизительное описание объекта, не позволит правильно моделировать его поведение. Слишком низкая степень абстракции сделает модель очень сложной, перегруженной деталями, и потому непригодной.

Для реализации такой абстракции в ООП существует понятие класса.

Формально **класс** - это шаблон поведения объектов определенного типа с заданными параметрами, определяющими состояние. Все экземпляры одного класса (объекты, порожденные от одного класса) имеют один и тот же набор свойств и общее поведение, то есть одинаково реагируют на одинаковые сообщения.

Говорят, что объект – это экземпляр класса. Говоря иначе, класс – это идея, а объект – её реализация.

**Поля (свойства)** класса - данные, которые хранит класс.

**Метод** класса – функция, объявленная внутри класса.

|  |
| --- |
| class Article  {  public $id;  public $title;  public $content;  // метод для вывода статьи  function view(){  echo "<h1>$this->title</h1><p>$this->content</p>";  }  }  $a = new Article();  $a->id = 1;  $a->title = 'Новая статья';  $a->content = 'Какой-то текст!';  $a->view(); |

В классе Article мы объявили три поля: id, title, content, а также один метод view, который

должен данную статью отображать на экране. Обратите внимание на новую конструкцию

$this->. Таким образом, мы получаем доступ к полям и методам изнутри класса. По сути,

this означает «этот», т.е. текущий экземпляр класса; new – ключевое слово, с помощью

которого создаётся объект. Затем мы обращаемся к его полям и заносим в них

определённые значения. После этого вызов метода view отобразит введённые данные на

экране.

Согласитесь, что задавать каждое поле класса в отдельности неудобно, поэтому был

изобретён конструктор.

**Конструктор класса** – метод, автоматически вызывающийся при создании экземпляра

класса.

|  |
| --- |
| class Article  {  public $id;  public $title;  public $content;  function \_\_construct($id, $title, $content){  $this->id = $id;  $this->title = $title;  $this->content = $content;  }  // метод для вывода статьи  function view(){  echo "<h1>$this->title</h1><p>$this->content</p>";  }  }  $a = new Article(1, 'Новая статья', 'Какой-то текст!');  $a->view(); |

Но вернёмся чуть выше и посмотрим, как же PHP реализует основные принципы ООП.

Пример полиморфизма (возможность объектов с одинаковой спецификацией иметь

различную реализацию.)

|  |
| --- |
| class A  {  function Test() { echo 'Это класс A<br />'; }  function Call() { $this->Test(); }  }  class B extends A  {  function Test() { echo 'Это класс B<br />'; }  }  $a = new A();  $b = new B();  $a->Call(); // выводит: «Это класс A»  $b->Test(); // выводит: «Это класс B»  $b->Call(); // выводит: «Это класс B» |

Инкапсуляция ограничивает доступ к составляющим компонентам объекта (свойствам и методам).

Каким образом происходит установка прав доступа? Для этого существует три специальных модификатора:

**public** позволяет обращаться к свойствам и методам отовсюду;

**private** позволяет обращаться к свойствам и методам только внутри текущего

класса;

**protected** позволяет обращаться к свойствам и методам только из текущего класса

и классов наследников

Например, следующий код вызовет ошибку:

|  |
| --- |
| class Article  {  private $id;  // …  }  $a = new Article();  echo $a->id; // Обратились к приватному полю не изнутри класса |

Разумеется, тут возникает вопрос в необходимости инкапсуляции. Зачем нам что-то от себя скрывать? На самом деле инкапсуляция имеет большой смысл при разработке крупных проектов,

когда присутствует совместная разработка. Например, представим следующую ситуацию: коллега по проекту прислал нам модель для работы с изображениями. Если она выполнена в процедурном подходе, либо без разграничения прав доступа на методы, то мы будем изучать её целиком, чтобы решить, какие из предоставленных возможностей использовать. В ООП же на все private-функции мы не станем обращать внимания, так как всё равно не сможем их вызвать. Данные функции используются внутри публичных методов этого же класса для решения каких-то прикладных задач.

Мы же извне обращаемся только к public функциям и совершенно не интересуемся внутренним устройством класса.

Константы и статические методы

Допустим, мы хотим создать класс с полезными математическими операциями:

|  |
| --- |
| class MathOperations  {  const PI = 3.14;  public function abs($x){  return ($x >= 0) ? $x : (-1) \* $x;  }  public function RangeLength($rad){  return 2 \* $rad \* self::PI;  }  } |

Константы класса задаются с помощью специального слова const. Обратите внимание,

что к константе мы обращаемся с помощью конструкции «self::», а не «$this->», так как

константа принадлежит классу, а не объекту.

Теперь внимательно посмотрим на получившийся класс. Сколько бы его экземпляров мы

не создали, все они будут похожи как две капли воды. Поэтому имеет смысл вообще избавиться от необходимости создавать объекты в данной ситуации.

**Статические** методы и свойства – это те, которые принадлежат классу, а не его

экземплярам.

|  |
| --- |
| class MathOperations  {  const PI = 3.14;  public static function abs($x){  return ($x >= 0) ? $x : (-1) \* $x;  }  public static function RangeLength($rad){  return 2 \* $rad \* self::PI;  }  } |

Обе функции мы сделали статическими. Это позволит обращаться к ним извне без создания экземпляра класса, например:

|  |
| --- |
| echo MathOperations::RangeLength(12); |

или

|  |
| --- |
| echo MathOperations::abs(-15); |

Обратиться извне к константе мы также можем, используя два двоеточия:

|  |
| --- |
| echo MathOperations::PI; |

Константа всегда принадлежит классу по той причине, что она неизменна и ничем бы не

отличались у экземпляров данного класса.

# Домашнее задание

1. Придумать класс, который описывает любую сущность из предметной области интернет-магазинов: продукт, ценник, посылка и т.п.

2. Описать свойства класса из п.1 (состояние)

3. Описать поведение класса из п.1 (методы).

4. Придумать наследников класса из п.1. Чем они будут отличаться?

5. Дан код

|  |
| --- |
| class A {  public function foo() {  static $x = 0;  echo ++$x;  }  }  $a1 = new A();  $a2 = new A();  $a1->foo();  $a2->foo();  $a1->foo();  $a2->foo(); |

Что он выведет на каждом шаге? Почему?

6. Немного изменим п.5

|  |
| --- |
| class A {  public function foo() {  static $x = 0;  echo ++$x;  }  }  class B extends A {  }  $a1 = new A();  $b1 = new B();  $a1->foo();  $b1->foo();  $a1->foo();  $b1->foo(); |

Объясните результаты в этом случае.

# Дополнительные материалы

1. Мэтт Вайсфельд – «Объектно-ориентированное мышление»
2. Андрей Шевченко – «Краткий сборник возможных вопросов и ответов на собеседовании.»