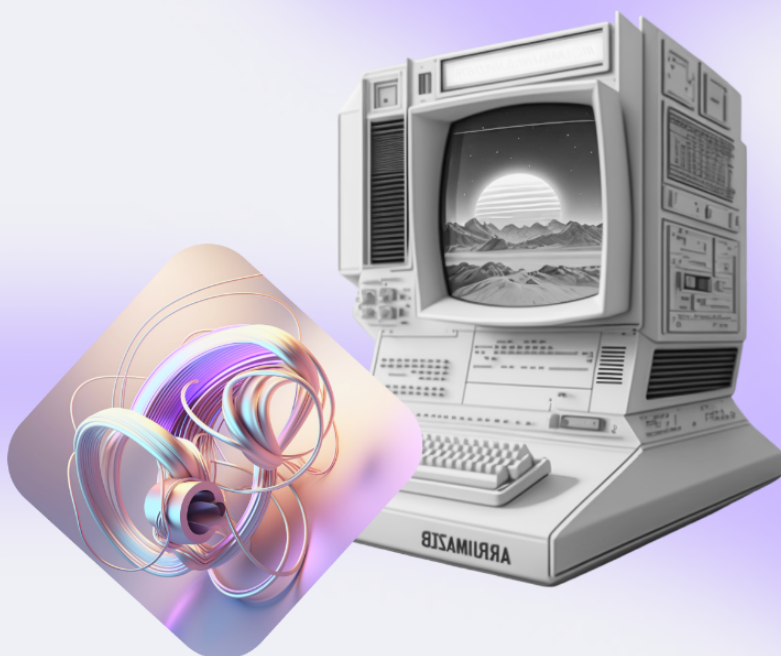


ООП

ОСНОВЫ PHP



Оглавление

Введение	3
Словарь терминов	4
Реализация основных концепций	5
Свойства	6
Методы	8
Конструктор	9
Области видимости и модификаторы доступа	10
Константы	11
Уровни доступа	12
Реализация основ ООП	13
Наследование	13
Инкапсуляция	16
Полиморфизм	17
Позднее статическое связывание	18
Абстрактные классы и интерфейсы	19
Пространства имен	22
Принципы написания ООП кода	23
Заключение	29
Домашнее задание	29
Что можно почитать еще?	30

Введение

Поскольку мы уже создали наше первое приложение, у нас появилась возможность увидеть, что даже не очень сложные информационные системы требуют структурного подхода к работе над ними для того, чтобы в будущем их можно было эффективно развивать и поддерживать.

И в этом отношении парадигма объектно-ориентированного программирования подходит как нельзя кстати, ведь объекты будут отражать логические сущности гораздо четче. И их не надо будет пытаться описывать в виде функций и скалярных переменных.

Вы наверняка помните, что объектно-ориентированное программирование (ООП) – это парадигма программирования, которая базируется на использовании объектов и классов. Преимущества ООП включают в себя:

1. Модульность: классы и объекты позволяют создавать модули, которые могут быть переиспользованы в различных частях программы. Это упрощает процесс разработки, тестирования и обслуживания приложений.
2. Инкапсуляция: классы и объекты позволяют скрыть сложность и детали реализации от других частей программы. Это обеспечивает более безопасную работу программы и упрощает ее модификацию.
3. Наследование: классы могут наследовать свойства и методы других классов, что упрощает разработку программ и позволяет создавать более эффективный и удобный код.
4. Полиморфизм: объекты могут быть использованы в различных контекстах и выполнять различные функции. Это позволяет создавать гибкие и расширяемые приложения, которые могут быть легко адаптированы к изменяющимся требованиям.
5. Разделение обязанностей: каждый объект выполняет определенную функцию, что позволяет легко распределять обязанности между членами команды разработчиков.

В целом, ООП упрощает разработку, тестирование и обслуживание программ, обеспечивает более безопасную и эффективную работу приложений, а также увеличивает возможности повторного использования кода.

Поэтому давайте погружаться в реализацию ООП в языке программирования PHP.

Словарь терминов

Объекты – это экземпляры классов, созданные согласно описанию самого класса. Они создаются с помощью оператора `new`.

Свойства – это переменные, которые принадлежат либо конкретному экземпляру класса, либо самому классу.

Метод – это функция, которая объявлена внутри класса.

Конструктор – это специальный магический метод, который вызывается при создании объекта класса, то есть каждый раз, когда в коде вызова употребляется ключевое слово `new`.

Область видимости – в PHP определяет область, в которой переменная или функция могут быть использованы.

Статическая область видимости – это означает, что переменная сохраняет свое значение между вызовами функции.

Константы – значения, которые не могут быть изменены во время выполнения скрипта. Они объявляются с помощью ключевого слова `const` или функции `define()`.

Инкапсуляция – это концепция, при которой данные и методы класса скрыты от внешнего мира, и доступ к ним осуществляется только через определенные методы класса. Это позволяет защитить данные от изменения случайным образом и обеспечивает более легкую поддержку кода.

Наследование – это механизм, позволяющий создавать новый класс на основе существующего, заимствуя его свойства и методы.

Подкласс – класс, который наследует свойства и методы другого класса.

Суперкласс – класс, от которого наследуются свойства и методы.

Полиморфизм – это способность объектов использовать один и тот же интерфейс для разных целей. Это позволяет создавать гибкий код, который может работать с различными типами данных и объектов, не зная об этом заранее.

parent – ключевое слово в PHP используется для обращения к методам и свойствам родительского класса из дочернего класса.

Абстракция – определением общих сущностей, которые могут быть реализованы в различных контекст

Абстрактный класс – это класс, который не может быть создан явно (то есть, через ключевое слово new).

Интерфейс – это схема, которая описывает набор методов и свойств, которые класс должен реализовать.

Пространства имен (namespaces) в PHP позволяют организовывать код в логически связанные группы и предотвращают конфликты имен между различными частями кода.

Реализация основных концепций

Классы являются основным элементом ООП в PHP. Классы определяют свойства и методы, которые будут использоваться объектами класса. Определение класса начинается с ключевого слова class, за которым следует имя класса и фигурные скобки, в которых определяются свойства и методы класса.

Рассмотрим пример определения класса:

```
class Student {  
    public string $name;  
    public int $age;  
  
    function __construct($name, $age) {  
        $this->name = $name;  
        $this->age = $age;  
    }  
  
    function sayHello(): string {  
        return "Привет, меня зовут {$this->name} и мне {$this->age} лет.";  
    }  
}
```

В этом примере мы определили класс Student с двумя свойствами - name и age, конструктором и методом sayHello(). Обратите внимание на то, что мы пока не создали ни одного экземпляра класса (объекта). Класс – это описание схемы, по которой будет создаваться конкретный экземпляр. Это очень похоже на Docker-образ и запущенный на его основе контейнер.

Например, мы создадим такой объект

```
$person = new Student("Иван", 18);
```



Объекты – это экземпляры классов, созданные согласно описанию самого класса. Они создаются с помощью оператора `new`.

Таким образом, мы создали объект с конкретными свойствами имени и возраста.

Теперь разберем каждый шаг подробнее.

Свойства



Свойства – это переменные, которые принадлежат либо конкретному экземпляру класса, либо самому классу.

Они определяются внутри класса.

Свойства, которые определили мы в примере выше – это динамические свойства. И они будут принадлежать конкретному экземпляру класса – объекту!

И они могут использоваться для хранения данных, которые относятся именно к этому объекту. Например, если у вас есть класс "Студент", свойства объекта могут хранить данные, такие как имя, возраст, оценки и т.д. для каждого студента.

Для доступа к динамическому свойству объекта используется имя переменной, хранящей объект, за которым следует стрелка и имя свойства. Например, если у вас есть объект "student" со свойством "name", вы можете получить доступ к свойству с помощью выражения

```
$student->name;
```

Значения свойств объекта могут быть изменены во время выполнения программы, что позволяет динамически изменять данные для каждого экземпляра класса в зависимости от текущих условий.

Например, если у студента выше случился день рождения, его возраст надо увеличить на 1:

```
$student->age++;
```



Свойства могут быть статическими. Это означает, что они принадлежат классу, а не его экземплярам. Такие свойства помечаются как **static**.

С точки зрения проектирования это означает, свойство не будет зависеть от персоналии.

Например, если мы проектируем класс студента, то можем зафиксировать, что любой студент имеет скидку на питание в столовой в размере 50%:

```
class Student {  
    public string $name;  
    public int $age;  
    public static float $discount = 0.5;  
  
    function __construct(string $name, int $age) {  
        $this->name = $name;  
        $this->age = $age;  
    }  
  
    function sayHello(): string {  
        return "Привет! Меня зовут {$this->name} и мне {$this->age} лет.";  
    }  
}
```

Для доступа к свойству класса используется имя класса, за которым следует двойное двоеточие и имя свойства:

```
echo Student::$discount;
```



Обратите внимание, что создавать экземпляр класса для обращения к такому свойству не требуется.

Важно отметить, что в 8 версии PHP для свойств можно указывать типы, которые они имеют. Строгая типизация свойств помогает избегать ошибок при разработке, упрощает тестирование и делает код более понятным. В нашем примере возраст будет иметь целочисленный тип, а скидка – тип числа с плавающей точкой.

Типы, которые можно указывать, могут быть как **скалярными** (строка, число, булевское значение), так и **другими классами**, так как в свойстве одного объекта вполне может быть сохранена ссылка на другой объект.



Динамические свойства определяют состояние конкретного объекта.

Свойство также может быть помечено как **«только для чтения»** при помощи ключевого слова **«readonly»**. Это можно делать, начиная с PHP 8.1. После того, как свойство получает значение, его больше нельзя переопределить.

💡 Такое ключевое слово может быть применено, например, для классов, которые отвечают за передачу данных между разными хранилищами. То есть, как только объект прочитан из одного хранилища, определенные его свойства уже нельзя будет поменять до передачи в другое.

Методы

Наравне с состоянием объект может иметь поведение. И поведение описывается методами.

💡 Метод в PHP – это функция, которая объявлена внутри класса.

Как и свойства, методы могут быть статическими и динамическими.

Динамические методы представляют собой действия или операции, которые могут быть выполнены на экземпляре класса.

Для объявления метода в PHP используется уже знакомое нам ключевое слово **"function"**, за которым следует имя метода и круглые скобки с возможными аргументами метода. Тело метода заключается в фигурные скобки и содержит инструкции, которые должны быть выполнены при вызове метода.

В нашем примере есть метод sayHello:

```
function sayHello(): string {  
    return "Привет, меня зовут {$this->name} и мне {$this->age} лет.";  
}
```

Обратите внимание на то, что как и в процедурном подходе, в ООП методы также могут иметь возвращаемый тип. Если метод принимает на вход какие-то значения, то для них также можно указывать ожидаемые типы.

Внутри себя динамический метод может обращаться к состоянию объекта и другим методам. Это делается при помощи ключевого слова **this**. Когда мы проектируем класс, мы не знаем, сколько объектов этого класса будет создано. Но внутри метода мы хотим уметь работать с состоянием какого-то конкретного объекта. Как видите, в методе sayHello через this мы обращаемся к имени и возрасту конкретного студента, который вызвал этот метод. Такая реализация позволяет нам делать вызовы следующего вида.

```
$students = [
```



```
new Student("Ольга", 20),  
new Student("Иван", 18)  
];  
  
foreach($students as $student) {  
    $student->sayHello();  
}
```

Обходя массив наших студентов, мы будем уверены, что метод `sayHello` будет обращаться в момент работы с объектом «Ольга» только к свойствам этого объекта, а не к свойствам объекта «Иван». То есть, мы получим вывод:

```
Привет, меня зовут Ольга и мне 20 лет.  
Привет, меня зовут Иван и мне 18 лет.
```

Статические методы аналогично статическим свойствам принадлежат классу, а не его экземплярам. Такие методы помечаются как **static**. Если продолжить пример со скидкой в столовой, то статическим может быть расчет скидки для студента. Вот как он будет выглядеть внутри класса:

```
public static function getDiscount(float $mealPrice): float {  
    return $mealPrice * Student::$discount;  
}
```

Если поведение метода не зависит от состояния объекта (значений полей класса), то такой метод обычно объявляют как статический.

Статические методы следует применять в двух случаях:

- Когда методу не требуется доступ к данным о состоянии объекта, поскольку все необходимые параметры задаются явно;
- Когда методу требуется доступ лишь к статическим полям класса.

Конструктор



Конструктор – это специальный магический метод, который вызывается при создании объекта класса, то есть каждый раз, когда в коде вызова употребляется ключевое слово `new`.

В нашем примере конструктор принимает два аргумента – `name` и `age` и устанавливает соответствующие значения свойств.

Наличие конструктора *необязательно* для класса. Но конструктор здорово помогает при необходимости задать начальное состояние объекта прямо в момент создания класса.

Начиная с PHP 8.0, параметры конструктора можно использовать для задания соответствующих свойств объекта. Это довольно распространённая практика — присваивать свойствам объекта параметры, переданные в конструктор, не производя никаких дополнительных преобразований. Определение свойств класса в конструкторе позволяет значительно сократить количество шаблонного кода для такого случая.

Пример выше можно будет переписать следующим образом:

```
class Student {  
    public function __construct(public string $name, public int $age) {  
    }  
}
```

Обратите внимание на то, что в данном случае переданные в конструктор параметры сразу присвоятся созданным свойствам. Однако такой тип записи не всегда очевиден и портит читаемость. Обязательно обсуждайте с командой разработки применение такого типа записи и его соответствие принятым в команде стандартам кода.

Области видимости и модификаторы доступа

Мы уже увидели в нашем примере класса Student употребление ключевого слова **public**. Это ничто иное, как указание уровня доступа свойств и методов в классе.



Помимо них есть также и область видимости — в PHP она определяет область, в которой переменная или функция могут быть использованы.

Из более ранних лекций мы помним, что есть глобальная и локальная области видимости.

Мы увидели ключевое слово **static**, которым помечали свойства и методы. Но обратите внимание на то, что вне контекста классов это ключевое слово обозначает **статическую область видимости**.



Статическая область видимости — это означает, что переменная сохраняет свое значение между вызовами функции.

Она объявляется с помощью ключевого слова `static`.

Например:

```
function my_function() {  
    static $static_variable = 0;  
    echo ++$static_variable; // значение увеличивается каждый раз, когда функция  
    вызывается  
}  
  
for($i = 0; $i < 5; $i++){  
    my_function();  
}
```

Как видите, переменная `static_variable` увеличилась ровно на количество итераций цикла, в котором вызывалась функция `my_function`.

Константы

Наравне со статическими переменными, которые сохраняют свое значение в контексте, есть еще и константы. Ведь в мире еще есть что-то неизменное. Например, число π , которое может использоваться для геометрических расчетов.

Также в качестве констант можно задавать и конфигурацию приложения, чтобы быть уверенными, что никто не сможет изменить её.



В PHP **константы** представляют собой значения, которые не могут быть изменены во время выполнения скрипта. Они объявляются с помощью ключевого слова `const` или функции `define()`.

Для объявления константы с помощью функции `define()` используется следующий синтаксис:

```
define('CONSTANT_NAME', 'value');
```

Такая константа имеет глобальную область видимости внутри скрипта. Константы могут быть определены как в глобальной области видимости, так и внутри классов. Если константа определена в классе, она может быть доступна только внутри этого класса и его наследников.

Это делается при помощи ключевого слова `const`:

```
const CONSTANT_NAME = 'value';
```

Для доступа к константе, определенной в классе, необходимо использовать имя класса и оператор двойного двоеточия (::):

```
echo MyClass::CONSTANT_NAME;
```

Уровни доступа

В контексте ООП в PHP выделяют три уровня доступа к свойствам и методам: общедоступный (public), защищенный (protected), закрытый (private).

Уже знакомый нам по примерам общедоступный уровень (public).



Общедоступный уровень (public) позволяет обращаться к свойствам и методам из любого места кода, включая области вне класса.

Для объявления общедоступного свойства или метода используется ключевое слово public. Любое свойство и метод, у которых явно не указан модификатор области видимости, считается public.

Так, мы можем обратиться, например, к полю \$name нашего класса Student как внутри класса, так и из другого класса или вообще любого места нашего кода.



Закрытый уровень (private) позволяет обращаться к свойствам и методам только внутри класса и нигде больше.

Для объявления закрытого свойства или метода используется ключевое слово private:

```
class Student {  
    private float $avgScore;  
}  
$student = new Student();  
echo $student->avgScore; // ошибка
```

Для того, чтобы разобраться с третьим типом – protected, перейдем к следующему разделу лекции.

Реализация основ ООП

В парадигме ООП существуют три ключевых понятия:

Инкапсуляция – это концепция, при которой данные и методы класса скрыты от внешнего мира, и доступ к ним осуществляется только через определенные методы класса. Это позволяет защитить данные от изменения случайным образом и обеспечивает более легкую поддержку кода.

Наследование – это механизм, позволяющий создавать новый класс на основе существующего, заимствуя его свойства и методы. Класс, который наследует свойства и методы другого класса, называется подклассом, а класс, от которого наследуются свойства и методы, называется суперклассом.

Полиморфизм – это способность объектов использовать один и тот же интерфейс для разных целей. Это позволяет создавать гибкий код, который может работать с различными типами данных и объектов, не зная об этом заранее. Например, в ООП полиморфизм может проявляться в перегрузке методов, виртуальных функциях, интерфейсах и т.д.

Давайте посмотрим на то, как они реализуются в языке PHP.

Наследование

В PHP наследование позволяет создавать новые классы, которые могут повторять или изменять свойства и методы родительских классов. Класс, от которого наследуются свойства и методы, называется родительским классом или суперклассом. Класс, который наследует свойства и методы другого класса, называется дочерним классом или подклассом.

Например:

```
class Person {  
    public string $name;  
    public array $access = [];
```

```

    public function __construct(string $name, array $access = []) {
        $this->name = $name;
        $this->access = $access;
    }

    public function checkAccess(string $room) {
        return in_array($room, $this->access);
    }
}

class Teacher extends Person {
    public function __construct(string $name) {
        parent::__construct($name, ['classroom', 'teachersroom']);
    }

    public function guideLecture() {
    }
}

class Student extends Person {
    public function __construct(string $name) {
        parent::__construct($name, ['classroom']);
    }
}

$student = new Student("Иван");
$teacher = new Teacher("Ольга");

var_dump($student->checkAccess('teachersroom')); // false
var_dump($teacher->checkAccess('teachersroom')); // true

```

В примере выше классы `Teacher` и `Student` наследуют свойства и методы класса `Person`. В дочерних классах мы переопределяем метод `__construct()` так, чтобы в нем задавалось только имя, а уровень доступа фиксировался. Затем мы вызываем родительский конструктор через ключевое слово `parent`.

При создании экземпляров дочерних классов, они автоматически наследуют метод `checkAccess()` из класса `Person`, и мы можем вызывать его, как если бы он был определен в классе `Teacher` или `Student`. При этом дочерний класс может иметь собственные методы, как, например, `guideLecture` у `Teacher`.



Мы уже увидели, что ключевое слово **parent** в PHP используется для обращения к методам и свойствам родительского класса из дочернего класса.

То есть, мы можем частично переопределить поведение метода, не переписывая его целиком.

Таким образом, наследование в PHP позволяет нам создавать иерархии классов, где каждый последующий класс может наследовать свойства и методы от предыдущего класса и дополнять их своими собственными свойствами и методами.

Но в нашем примере мы начали работать с доступами. И, как видите, они имеют открытый (public) доступ. То есть, любой пользователь кода сможет по собственному желанию поменять доступы, например, у студента. Нам это не очень нравится, так как в таком случае метод проверки `checkAccess` теряет свой смысл.

Тогда наше поле `$access` можно было бы сделать закрытым (private). Проблема тут в том, что в таком случае оно перестанет наследоваться у `Teacher` и `Student`. И здесь нам на помощь и придет третий модификатор доступа – `protected`. Свойство или метод, помеченные этим модификатором, будут наследоваться, но не будут доступны вне класса.

Отредактируем класс `Person` так, чтобы он стал безопаснее:

```
class Person {
    protected string $name;
    protected array $access = [];

    public function __construct(string $name, array $access = []) {
        $this->name = $name;
        $this->access = $access;
    }

    public function checkAccess(string $room) {
        return in_array($room, $this->access);
    }
}
```

При наследовании может возникнуть ситуация, когда мы не хотим, чтобы наследники переопределяли, например, поведение того или иного. Для таких случаев предусмотрено ключевое слово `final`. Им мы можем пометить метод `checkAccess`, чтобы ни `Teacher`, ни `Student` не могли влиять на его поведение.

При этом они смогут продолжать его вызывать:

```
public final function checkAccess(string $room) {
    return in_array($room, $this->access);
}
```

Инкапсуляция

Поскольку мы уже познакомились с модификаторами доступа к методам и свойствам классов, то уже понимаем, как реализуется скрывание доступов к сущностям в языке PHP. Но здесь возникает другая задача.

Если у нашего класса есть `protected` или `private` поля, к ним может потребоваться доступ извне. Например, какой-то другой класс захочет получить доступ к имени студента. Или же какой-то из классов будет менять поля в другом классе.

В такой ситуации общепринятой практикой является создание так называемых `getter` и `setter` методов. Например, для получения имени, которое теперь нельзя просто так напрямую считать из экземпляра нашего класса, нужно создать метод:

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

🔥 В случае же с заданием защищенного свойства нужно быть чуть аккуратнее. Ведь мы должны как-то валидировать создаваемое значение. Например, мы будем проверять задаваемое имя по списку, чтобы в нем не содержалось нецензурных выражений или чисел.

```
public function setName(string $name) : void {  
    if($this->validateName($name)) {  
        $this->name = $name;  
    }  
}
```

Таким образом, для защищенных (`protected` или `private`) свойств мы можем создавать пары методов:

- `get` – для чтения значения свойства;
- `set` – для задания значения свойства.

Полиморфизм

Часто полиморфизм называют «наследованием наоборот». И сейчас мы разберемся – почему.

Полиморфизм по своей сути говорит, что одна и та же строчка кода может вызывать методы разных классов (ведь при наследовании они могут переопределять свое поведение, не меняя имени), в зависимости от того, с каким объектом мы работаем в данный момент.

Рассмотрим пример, где в классе Person определим метод goToLunch():

```
public function goToLunch() : string {  
    return "Просто кто-то идет пообедать";  
}
```

Теперь переопределим этот метод в классах-наследниках. В Teacher:

```
public function goToLunch() : string {  
    return "Учитель любит ходить в диетическую столовую";  
}
```

И в Student:

```
public function goToLunch() : string {  
    return "Студенту важна экономия!";  
}
```

Создадим массив объектов:

```
$persons = [  
    new Person("Иван"),  
    new Student("Олег"),  
    new Teacher("Мария")  
];
```

Реализация полиморфизма на практике будет заключаться в том, что если мы обойдем этот массив циклом, вызывая у каждого элемента метод goToLunch, то движок PHP вызовет точно тот метод, который определен в классе, экземпляром которого является текущий элемент. То есть:

```
foreach ($persons as $person) {  
    echo $person->goToLunch();  
}
```



Здесь важно отметить, что стандартный массив в PHP не будет гарантировать то, что в него поместят только объекты нужного семейства.

Значит, это можно проверить в цикле. И снова, благодаря полиморфизму, мы можем проверить это просто. Ведь мы знаем, что семейство классов `Person` гарантированно имеет метод `goToLunch`. А значит, мы должны проверить, что текущий элемент внутри итерации является экземпляром класса `Person`. Полиморфизм и говорит нам о том, что и `Teacher`, и `Student` будут проходить эту проверку. Разумеется, в обратную сторону (сторону наследования) это не работает. Так, если мы ожидаем увидеть класс `Student`, а нам придет `Person`, то проверка не сработает.

Проверку принадлежности классу в PHP можно осуществить оператором `instanceof`:

```
foreach ($persons as $person) {  
    if($person instanceof Person){  
        echo $person->goToLunch();  
    }  
}
```

Позднее статическое связывание

Мы уже познакомились с указателями `this` и `parent` в ООП. Но помимо них есть еще пара важных ключевых слов. Это `self` и `static`.



Ключевое слово **`self`** при обращении к методу или свойству, ссылается на класс, в котором оно записано.



Ключевое слово **`static`** ссылается на класс, в котором был вызван метод или свойство.

Разберемся на примере. Допустим, в наше семейство `Person` будет добавлены статическое свойство, хранящее литературное название типа (например, у `Person` – «Персона», у `Student` – «Студент»), а также метод, который будет идентифицировать объект:

```
protected static string $writableName;  
  
public function whoAmI() : string {  
    return "Я - " . self::$writableName;  
}
```

Теперь, если мы вызовем этот метод, то увидим, что оба вызова вернули строку «Я – Персона», так как указатель `self` всегда ссылается на класс, в котором он записан:

```
$student = new Student("Иван");
```

```
$teacher = new Teacher("Ольга");  
  
var_dump($student->whoAmI());  
var_dump($teacher->whoAmI());
```

А у нас метод `whoAmI` определен именно в `Person`.

Для того, чтобы сработал эффект, называемый поздним статическим связыванием, нужно заменить `self` на `static` в методе `whoAmI`:

```
public function whoAmI() : string {  
    return "Я - " . static::$writableName;  
}
```

В таком случае аналогичный вызов вернет уже строки:

```
Я - Студент  
Я - Учитель
```

Это происходит потому, что `static` уже перебросит нас в конкретные классы, где свойство `writableName` переопределено.

Абстрактные классы и интерфейсы

В примерах выше мы использовали класс `Person`, который обобщает возможных конкретных персон, которые могут пользоваться нашим приложением – Учителя и Студента. Такое обобщение в программировании называется абстракцией.



Абстракция – определение общих сущностей, которые могут быть реализованы в различных контекстах.

Так Персона имеет общие черты и для Учителя, и для Студента. Но в реализациях они, как мы помним, отличаются.

По сути, нам не нужно непосредственно создавать экземпляры класса Персона – нас больше интересуют конкретные представители. Ведь мы хотим знать, сможет ли отдельно взятый объект выполнить то или иное действие. Но обобщение помогает нам не писать несколько раз один и тот же код.

Для удобства в PHP созданы абстрактные классы.



Абстрактный класс – это класс, который не может быть создан явно (то есть, через ключевое слово `new`). Также он может содержать как методы с явной

реализацией, так и абстрактные методы, которые не имеют реализации в самом классе, но должны быть реализованы в классах-наследниках.

Абстрактные классы могут быть полезны, когда требуется определить общие свойства и методы для группы классов, но не требуется создавать экземпляры этого класса напрямую.

Возвращаясь к классу `Person`, мы можем сказать, что метод `goToLunch` в нем не очень-то нужен с точки зрения реализации. Давайте модифицируем код. Абстрактные классы, методы и свойства в PHP помечаются ключевым словом `abstract`:

```
abstract class Person {
    protected string $name;
    protected array $access = [];

    public function __construct(string $name, array $access = []) {
        $this->name = $name;
        $this->access = $access;
    }

    public function checkAccess(string $room) {
        return in_array($room, $this->access);
    }

    abstract public function goToLunch();
}
```

Как видите, у метода теперь нет тела. Но если мы наследуемся от этого класса, то в случае, если мы не реализуем этот метод (то есть, не опишем его явно), PHP сгенерирует ошибку о том, что мы нарушаем соглашение с абстрактным методом.

Но что, если мы рассматриваем в нашем приложении не только Персон, но и, например, животных? Ведь животное тоже иногда хочет пойти поесть. Наследоваться абстрактного класса `Person` тут не очень правильно – ведь тут речь про человека. Но мы хотим, чтобы был контракт, который гарантирует, что у класса есть метод `goToLunch`.

Для таких вещей пригодятся интерфейсы.



Интерфейс – это схема, которая описывает набор методов и свойств, которые класс должен реализовать.

Интерфейс определяет только имена методов, но не их реализацию. Классы, которые реализуют интерфейс, должны определить реализацию всех методов, определенных в интерфейсе.

При именовании интерфейсов стоит придерживаться следующих общепринятых подходов.

1. Имя интерфейса должно начинаться с заглавной буквы, а каждое последующее слово в имени интерфейса должно начинаться с заглавной буквы. Например, `MyInterface`, `AnotherInterface`.
2. Имя интерфейса должно являться существительным, описывающим общий набор методов и свойств, которые должны быть реализованы в классе, который реализует этот интерфейс
3. Имя интерфейса должно отражать его функциональное назначение. Например, если интерфейс определяет методы для работы с базой данных, то его имя должно содержать слова, связанные с базой данных, например, `DatabaseInterface`.
4. Если интерфейс определяет методы, которые должны быть реализованы в классах, имеющих общую цель или назначение, то имя интерфейса должно отражать эту общую цель. Например, если интерфейс определяет методы для работы с виджетами на веб-странице, то его имя может содержать слово `Widget`, например, `WidgetInterface`.

Примеры правильных имен интерфейсов: `Iterator`, `Countable`, `DatabaseInterface`, `WidgetInterface`.

У нас есть потребность в контракте, который говорит, что класс умеет ходить за едой. Назовем его `AbleToLunch`:

```
interface AbleToLunch {  
    public function goToLunch();  
}
```

Теперь код, например, класса `Student` будет выглядеть так:

```
class Student implements AbleToLunch {  
}
```

Интерфейсы удобны, когда нам надо гарантировать наличие у группы классов ожидаемого поведения. Благодаря полиморфизму, мы можем проверять

соответствие контракту все тем же `instanceof`, но теперь указывая не конкретный класс, а интерфейс.

Абстрактные классы пригодны для определения групп классов, в которых будут общие методы с одинаковым поведением, но каждый из классов будет иметь свои реализации некоторых абстрактных методов. Таким образом, абстрактные классы больше подходят для тесно связанных семейств классов.

Интерфейсы же гарантируют соблюдение наличия в группе классов определенных методов, но без аспектов реализации. Поэтому интерфейсы подходят для групп более разрозненных по своему смыслу классов.

Пространства имен

Как вы уже могли догадаться, в именовании классов больших ограничений нет. Но при этом может произойти ситуация, когда кто-то решит назвать свой класс также, как ваш.

Рассмотрим пример. Мы написали в прошлом уроке приложение, которое обрабатывает запросы к файловому хранилищу. У нас может быть хранение в CSV, XML и других форматах.

Для обработки каждого формата надо будет создать классы типа:

- `CsvStorage`
- `XmlStorage`
- `JsonStorage`

Но эту же задачу можно решить изящнее. Более того, это решение пригодится нам в следующей лекции, где мы будем собирать наше новое приложение.

Мы можем условиться, что любой обработчик будет храниться в классе `Storage`. Но система папок будет следующей:

- `StorageProviders`
 - `Csv`
 - `Storage`
 - `Xml`
 - `Storage`

- Json
 - Storage

Это можно удобно отразить в пространстве имен.



Пространства имен (namespaces) в PHP позволяют организовывать код в логически связанные группы и предотвращают конфликты имен между различными частями кода.

Пространство имен определяется ключевым словом namespace, за которым следует имя пространства имен:

```
namespace StorageProviders\Csv;  
class Storage {  
}
```

Или же:

```
namespace StorageProviders\Xml;  
class Storage {  
}
```

Если мы хотим использовать класс из другого пространства имен, то его имя можно задать с помощью оператора use.

Например:

```
use StorageProviders\Csv;  
$obj = new Storage();
```

Как вы уже наверняка догадались, в конвенциях PSR пространство имен часто соответствует папке, в которой лежит класс (но необязательно). То есть, по пространству имен мы можем определить, где же искать исходный код класса. И на этом в ООП строится уже знакомая нам автозагрузка файлов с классами, с которой мы познакомимся в следующей лекции.

Принципы написания ООП кода

Наиболее распространенной ошибкой в создании ООП-приложений при переходе от процедурного кода к объектному является повсеместное использование статических методов и, как следствие свойств. То есть, функция:

```
function readConfig(string $configAddress): array|false{  
    return parse_ini_file($configAddress, true);  
}
```

Просто превращается в метод:

```
public static function readConfig(string $configAddress): array|false{  
    return parse_ini_file($configAddress, true);  
}
```

🔥 Но в чем же тут проблема? Дело в том, что нам придётся каким-то образом обеспечивать как доступность этого метода во всём приложении, так и возможность передачи в него адреса конфигурации. Это крайне неудобно, а также нарушает принцип инкапсуляции, делая метод по сути глобальным.

Работая с ООП нужно менять свой взгляд на проектирование кода. Отлично описаны подобные подходы в книге Мэтта Вайсфельда “Объектно-ориентированное мышление”. Но мы постараемся вкратце сформировать их для себя.

Работая с приложением до этого момента, мы писали функции, которые выполнялись в процессе работы. Например, “читать конфигурацию”, “вывести всех пользователей” и другие. Мы думали действиями. При этом данные, которые передавались между этими действиями, отходили на второй план – мы были вынуждены подстраивать их поведение под создаваемые действия. Действительно – создать функцию часто быстрее, чем описывать классы объекты и продумывать их взаимодействие. Но с ростом объёма кода хитросплетения функций становятся всё труднее управляемыми. Всё это делает наше приложение переусложнённым.

Не стоит, однако, думать, что процедурный подход в программировании плох – у него есть свои области применения, но PHP ООП язык, и заточен на то, чтобы сложные приложения создавались именно на базе объектной парадигмы.

Когда мы говорим об ООП, мы должны думать объектами. Все наши программы, которые мы напишем с применением ООП подхода, будут состоять из объектов со своими данными и набором доступных действий.

На примере одного только метода `readConfig` мы сейчас разберем, как же ООП поможет нам в построении приложения на нашей следующей лекции.

Вы наверняка заметили, что концепция единой точки входа в приложение крайне удобна. Действительно, не стоит от неё отказываться.

Давайте выполним команду, чтобы создать объектный автозагрузчик. Пока мы не будем погружаться в его работу – мы разберем это на следующей лекции. Он работает очень похоже на файловый автозагрузчик, только полагаться будет на пространства имен:

```
docker run --rm -v $(pwd)/php-cli/code:/code php-gb-cli composer init
```

Зададим имя нашего приложения:

```
app/oop
```

В процессе работы Composer создаст для нас пространство имен, которое будет смотреть в директорию src.

Создадим точку входа – файл app.php в корне проекта. Пока разместим там только наш автозагрузчик:

```
<?php  
require('./vendor/autoload.php');
```

Теперь в папке src мы создадим класс App. У него, согласно нашим установкам в начале, будет пространство имен App\Oop:

```
<?php  
  
namespace App\Oop;  
  
class App {  
  
}
```


Мы хотим работать с нашим приложением. Приложение – это объект. У него есть свойства и поведение. В самом простом варианте, что это будут за свойства и методы?



1. Свойство конфигурации
2. Метод запуска приложения

Конфигурацию и обращение к ней можно описать прямо внутри App. Но это неправильно. Конфигурация – это тоже объект! Для неё нужен свой класс. Что он будет делать? Основной его задачей будет чтение файлов и раскладывание в массив настроек приложения.

Классов становится больше, поэтому неплохо бы их визуализировать при помощи схемы. Можно использовать здесь упрощенную UML нотацию.

 UML (Unified Modeling Language) – это стандартизированный язык визуального моделирования, который используется для описания, визуализации, проектирования и документирования различных аспектов программных систем.

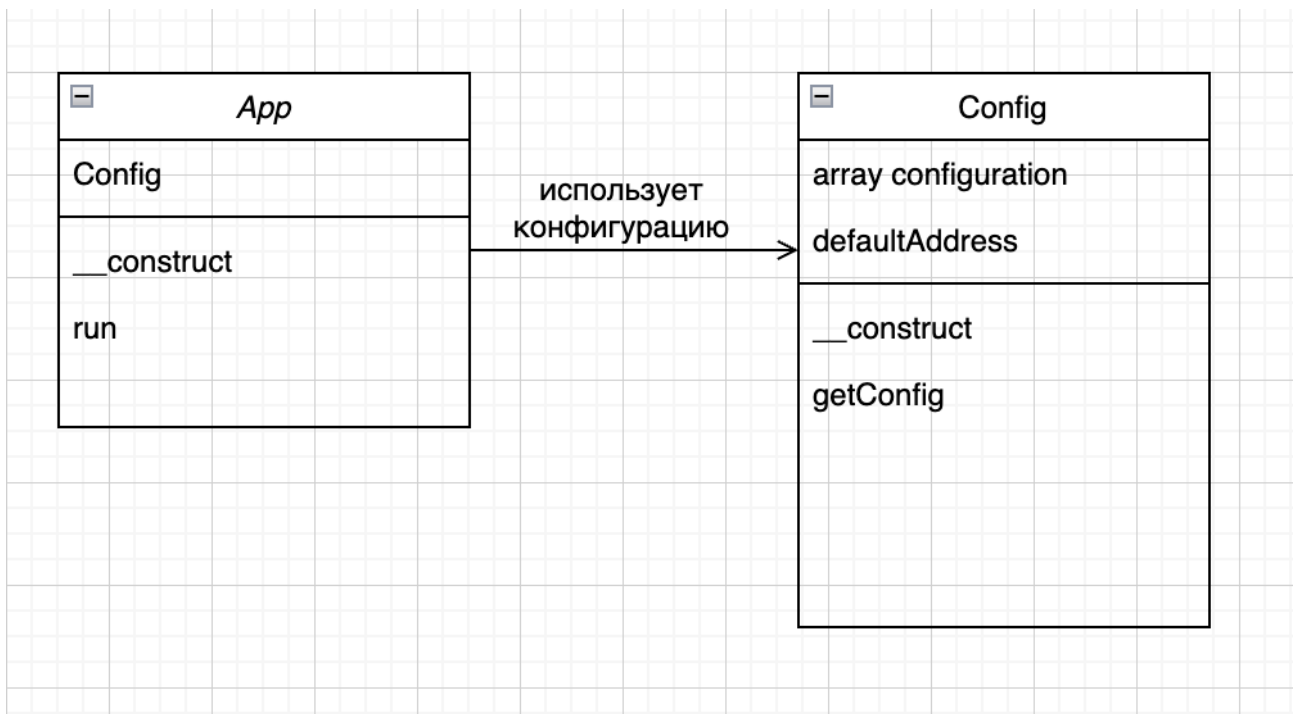
Он предоставляет нотацию для описания классов, объектов, компонентов, взаимодействий между ними и других аспектов системы.

Для описания классов в UML используется диаграмма классов, которая включает следующие основные элементы:

- Классы: классы представляют собой абстракции объектов, которые имеют общие свойства и поведение. Они представляются в диаграмме классов прямоугольниками, в которых указывается имя класса.
- Атрибуты: атрибуты – это свойства или данные, которые принадлежат классу. Они представляются внутри класса в виде списка с указанием их имен и типов данных.
- Методы: методы – это поведение класса, то есть операции или функции, которые класс может выполнять. Они также представляются внутри класса и обычно снабжены списком параметров и типом возвращаемого значения.
- Модификаторы видимости: модификаторы видимости определяют доступность атрибутов и методов класса для других классов. Они могут быть следующими:
 - + (public) – доступ открыт для всех.
 - - (private) – доступ ограничен только к текущему классу.
 - # (protected) – доступ разрешен для текущего класса и его подклассов.

- ~ (package) – доступ ограничен в пределах текущего пакета.
- Отношения между классами: на диаграмме классов также отображаются отношения между различными классами. Наиболее распространенные отношения это наследование (стрелка с пустым треугольником) и ассоциация (обычная стрелка). Есть также агрегация (стрелка с пустым ромбом) и композиция (стрелка с заполненным ромбом).
- Абстрактные классы и интерфейсы: UML позволяет указывать абстрактные классы (абстрактные методы или атрибуты) и интерфейсы, которые используются для определения общего поведения и стандартных контрактов.

Давайте нарисуем схему для нашего простого приложения. Для этого можно использовать браузерное [приложение](#).



Итак, в объекте конфигурации мы можем хранить адрес конфигурации по-умолчанию, саму конфигурацию, а также предоставлять к конфигурации доступ извне (чтобы никто не мог поменять её в процессе работы приложения).

Теперь давайте опишем саму логику класса конфигурации:

```
namespace App\Oop;

class Config {

    private const DEFAULT_ADDRESS = "config.ini";

    private array $configuration = [];

    public function __construct(string $address = null)
    {
        if($address == null || empty($address)){
            $address = Config::DEFAULT_ADDRESS;
        }

        $this->configuration = parse_ini_file(
            __DIR__ . '/../' . $address,
            true
        );
    }

    public function get() : array {
        return $this->configuration;
    }
}
```

По сути, мы несильно поменяли само поведение получения конфигурации. Но теперь она хранится внутри объекта. Что это означает? Теперь мы можем работать с состоянием!

```
namespace App\Oop;

use App\Oop\Config;

class App {

    public static Config $config;

    public function __construct()
    {
        self::$config = new Config();
    }
}
```

То есть, мы создаём статическое свойство внутри App, к которому можно обращаться в любой точке приложения. Мы при этом не нарушаем инкапсуляции, так как сама конфигурация приложения хранится внутри класса. Как мы можем к ней обращаться? Давайте создадим файл конфигурации и метод run внутри нашего приложения.

```
[storage]
address=birthday.txt
```

Теперь мы можем в любой точке приложения обращаться к конфигурации следующим образом:

```
public function run() {  
    return App::$config->get()['storage']['address'];  
}
```

Именно такими методами следует работать и думать, когда мы говорим о программировании в рамках ООП.

Заключение

В этой лекции мы начали знакомство с объектно-ориентированным программированием, чтобы в следующем занятии плотно погрузиться в применение классов и объектов и написать новое приложение.

Домашнее задание

1. Придумайте класс, который описывает любую сущность из предметной области библиотеки: книга, шкаф, комната и т.п.
2. Опишите свойства классов из п.1 (состояние).
3. Опишите поведение классов из п.1 (методы).
4. Придумайте наследников классов из п.1. Чем они будут отличаться?
5. Создайте структуру классов ведения книжной номенклатуры.
 - Есть абстрактная книга.
 - Есть цифровая книга, бумажная книга.
 - У каждой книги есть метод получения на руки.

У цифровой книги надо вернуть ссылку на скачивание, а у физической – адрес библиотеки, где ее можно получить. У всех книг формируется в конечном итоге статистика по кол-ву прочтений.

Что можно вынести в абстрактный класс, а что надо унаследовать?

6. 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?

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();
```

Что он выведет теперь?

Что можно почитать еще?

1. <https://phptherightway.com/>
2. <https://php.net>
3. РНР 8 | Котеров Дмитрий Владимирович
4. Объектно-ориентированное мышление | Мэтт Вайсфельд

5. <https://stackoverflow.com/questions/34034730/how-to-enable-color-for-php-cli>
- как покрасить вывод консоли
6. <https://phptoday.ru/post/gotovim-lokalnuyu-sredu-docker-dlya-razrabotki-na-php>
- варианты образа Composer