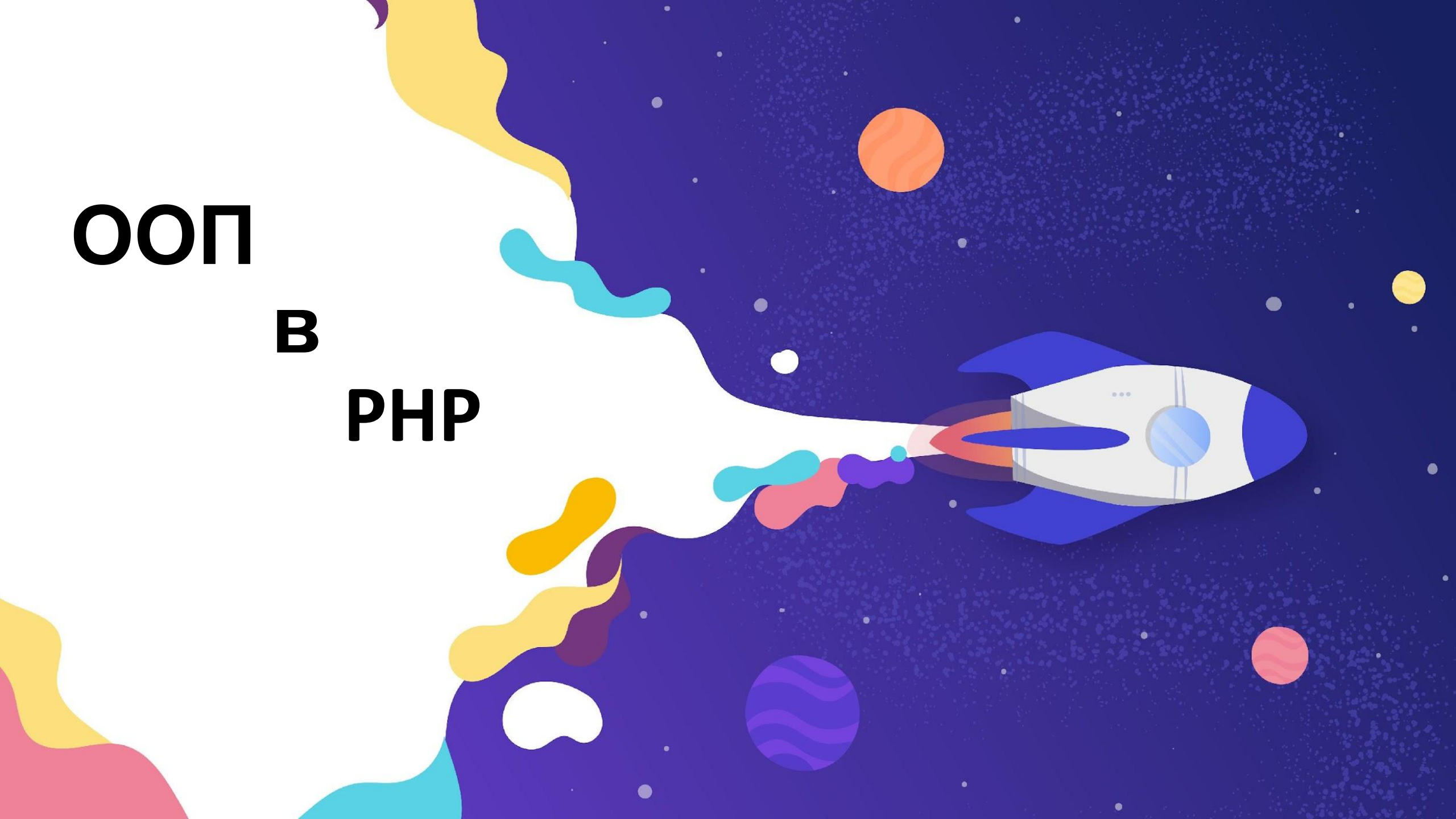
ООП

в

PHP



ООП. Общая концепция

ООП – подход к написанию утилит в виде моделирования информационных элементов. Предусматривает быструю разработку сложного контента. Утилиты, в основе которой лежит соответствующая концепция, обладают хорошей масштабируемостью и легкой поддержкой на протяжении длительного времени.

ООП в PHP появилось относительно недавно – после 5 версии. До этого момента соответствующая концепция не поддерживалась. Но классы и объекты здесь встречаются с 3 версии.

ООП. Общая концепция

Объектно-ориентирующая составляющая:

- помогает исправлять фатальные ошибки языка;
- способствует более простому и быстрому пониманию кодификации;
- минимизирует избыточность информации;
- позволяет управлять имеющимися моделями предельно эффективно.

Она помогает воссоздавать новые игры и сложные программы, задействовать базы данных, а также быстро осваиваться в коддинге и обнаруживать ошибки софта.

ООП. Общая концепция

Объектно-ориентирующая составляющая:

- помогает исправлять фатальные ошибки языка;
- способствует более простому и быстрому пониманию кодификации;
- минимизирует избыточность информации;
- позволяет управлять имеющимися моделями предельно эффективно.

Она помогает воссоздавать новые игры и сложные программы, задействовать базы данных, а также быстро осваиваться в коддинге и обнаруживать ошибки софта.

ООП. Классы

Класс – своеобразный шаблон, который позволяет представлять реальное понятие. Отвечает за управление свойствами задействованного элемента. Позволяет манипулировать, используя разного рода методы.

Класс подразумевает объединение нескольких «себеподобных» в иерархию наследования.

Свойство класса – переменная, которая задействована для хранения данных об объекте.

Метод класса – функция, которая отвечает за выполнение тех или иных действий. Последние имеют тесную связь с объектами.

ООП. Объекты

Для того, чтобы задействовать класс, требуется инстанцировать его. Конечный результат — это и есть объект. Реальный элемент, который предусматривает возможность работы над ним, обладающий конкретными свойствами.

Создание объекта происходит посредством вызова конструктора класса.

Конструктор - это особый метод, в котором можно задать функционал, который будет выполнен при инициализации объекта.

Объявление конструктора является необязательным. В случае отсутствия явно заданного метода в классе, конструктор будет вызван неявно.

ООП. Конструктор и деструктор

В конструктор можно передавать аргументы. Обычно переданные аргументы используются для того, чтобы установить свойства объекта.

```php
class Foo
{
    private int $bar;

    public function __construct(int $bar)
    {
        $this->bar = $bar;
    }
}
```

ООП. Конструктор и деструктор

Однако в PHP 8.0 появилась возможность сокращенного синтаксиса установки свойств без их явного объявления

```php
class Foo
{
    public function __construct(private int $bar)
    {}
}

$foo = new Foo(10);
```

ООП. Конструктор и деструктор

```php
class Foo
{
    public function __construct(private int $bar)
    {}

    public function __destruct()
    {
        // disconnect some service
    }
}
```

Как правило, задание деструктора в явном виде в классе не является обязательным. При завершении времени жизни объекта деструктор будет вызван не явным образом и сборщик мусора очистит память объекта.

Деструктор обычно используется, если при выгрузке объекта из памяти, необходимо например произвести какие-то действия со сторонними сервисами.

ООП. Модификаторы доступа

С помощью специальных модификаторов можно задать область видимости для свойств и методов класса. В PHP есть три таких модификатора:

public: к свойствам и методам, объявленным с данным модификатором, можно обращаться из внешнего кода и из любой части программы

protected: свойства и методы с данным модификатором доступны из текущего класса, а также из классов-наследников

private: свойства и методы с данным модификатором доступны только из текущего класса

ООП. Константы

Константы, как и переменные хранят определенное значение, только в отличие от переменных значение констант может быть установлено только один раз, и далее его изменить уже невозможно.
Константы обычно определяются для хранения значений, которые должны оставаться неизменными на протяжении всей работы скрипта.

Для определения константы применяется оператор const, при этом в названии константы знак доллара $ (в отличие от переменных) не используется.
Для констант, как и для свойств и методов актуально использование модификаторов доступа. Именовать констант принято заглавными латинскими буквами. В случае, если имя константы должно состоять из нескольких слов, они разделяются знаком “_”

```
class Cat extends Animal
{
    protected const PET_NAME = 'Cat';
}
```

ООП. Статические методы и свойства

Статическое свойство или метод не принадлежит какому-то объекту, а классу в целом. Для их использования не нужно создать объект.

Вызов статических методов осуществляется из самого класса:

```
class Profile
{
    private static float $retirementAge;

    public static function getRetirementAge(): float
    {
        return self::$retirementAge;
    }
}
echo Profile::getRetirementAge();
```

ООП. Позднее статическое связывание

Позднее Статическое Связывание (Late Static Binding, LSB) позволяет объектам все также наследовать методы у родительских классов, но помимо этого дает возможность унаследованным методам иметь доступ к статическим константам, методам и свойствам класса потомка, а не только родительского класса.

```php
class Animal
{
    protected const NAME = 'Animal';

    public static function getName(): string
    {
        return self::NAME;
    }

    public static function getStaticName(): string
    {
        return static::NAME;
    }
}
```

```php
class Cat extends Animal
{
    protected const NAME = 'Cat';
}

echo Cat::getName();
echo Cat::getStaticName();
```

Результат:
Animal
Cat

ООП. Инкапсуляция

Инкапсуляция – важная составляющая объектно-ориентированного программирования. С ее помощью удается ограничивать доступ к тем или иным свойствам/методам объекта.

Суть инкапсуляции состоит в названии. In capsula (в капсуле), т.е. изолировано от внешнего взаимодействия.

Инкапсуляция служит для того, чтобы отделить функционал класса от данных, сокрыть тот функционал, который относится исключительно к данному классу и открыть те функции для внешнего взаимодействия, которые необходимы.

Как правило, говоря об инкапсуляции, имеется в виду инкапсуляция методов, а свойства по-умолчанию должны быть инкапсулированы, а для доступа к ним должны быть использованы специальные методы.

ООП. Геттеры и сеттеры

Верным подходом к разработке считается использование специальных методов для взаимодействия со свойствами - геттеров и сеттеров.

```
class User
{
    private string $name;

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

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

ООП. Геттеры и сеттеры

Конструктор также является своеобразным сеттером. Верный подход заключается в том, чтобы все свойства, у которых нет значения по-умолчанию, задавались в конструкторе, а для остальных использовались сеттеры.

```php
class User
{
    private ?string $patronymic = null;

    public function __construct(
        private string $firstName
    ) {}

    public function getFirstName(): string
    {
        return $this->firstName;
    }

    public function setPatronymic(string $patronymic): void
    {
        $this->patronymic = $patronymic;
    }

    public function getPatronymic(): string
    {
        return $this->patronymic;
    }
}
```

ООП. Пространство имен

Пространства имен введены в PHP для решения проблем в больших PHP-библиотеках. В PHP все определения классов глобальны, поэтому авторы библиотек должны выбирать уникальные имена для создаваемых ими классов.

Это делается для того, чтобы при использовании библиотеки совместно с другими библиотеками не возникало конфликтов имен. Обычно это достигается введением в имена классов префиксов. Например: при использовании класса **dataBase** - велика вероятность, что такое имя класса будет присутствовать и в других библиотеках, а при их совместном использовании возникнет ошибка. Одним из решений является использование другого имени для класса.
Например: **myLibraryDataBase** Такие действия приводят к чрезмерному увеличению длины имен классов.

ООП. Пространство имен

Пространства имен позволяют разработчику управлять зонами видимости имен, что избавляет от необходимости использования префиксов и чрезмерно длинных имен. Все это служит повышению читабельности кода.

Пространства имён доступны в PHP начиная с версии 5.3.0.

Пространство имён определяется посредством ключевого слова *namespace*, которое должно находиться в самом начале файла.

ООП. Пространство имен

```php
namespace MyProject\MyFolder;

class MyAwesomeClass
{
    public function myMethod(): string
    {
        return 'some value';
    }
}
…
$object = new \MyProject\MyFolder\MyAwesomeClass();
echo $object->myMethod();
```

Это пространство имен может быть использовано в разных файлах.

Пространства имен могут включать определения классов, констант и функций.

Но не должны включать обычного кода.

ООП. Пространство имен

Поиск неквалифицированного имени класса (т.е. не содержащего **::**) осуществляется в следующей последовательности:

1. Попытка найти класс в текущем пространстве имен (т.е. префиксирование класса именем текущего пространства имен) без попытки автозагрузки (autoload).
2. Попытка найти класс в глобальном пространстве имен без попытки автозагрузки (autoload).
3. Попытка автозагрузки в текущем пространстве имен.
4. В случае неудачи предыдущего - отказ.

ООП. Пространство имен. Оператор use

Создание объектов классов, у файлов которых достаточно глубокая вложенность, может выглядеть достаточно громоздко в коде и делать его плохо читабельным. Для избежания подобных проблем существует оператор use, который позволяет указывать namespace классов в определенно заданном месте и тем самым улучшает читабельность кода. Сравним:

```php
function aaa():
\MyProject\Application\Services\MyAwesomeService\Singleton
{
    $a = new
\MyProject\Application\Services\MyAwesomeService\Singleton();
    $a->doSomething();

    return $a;
}
```

```php
use
\MyProject\Application\Services\MyAwesomeService\Singleton;

function aaa(): Singleton
{
    $a = new Singleton();
    $a->doSomething();

    return $a;
}
```

ООП. Автозагрузка

В правильно выстроенной архитектуре каждый класс должен располагаться в отдельном файле. В таком случае необходимо каждый включать данный файл в место его использования.
При работе с несколькими классами в одном модуле подключение всех необходимых файлов является достаточно затратным и затруднительным для контроля.

Как правило, включение классов решается с помощью механизма автозагрузки. Работает механизм следующим образом, с помощью функции spl_autoload_register.

Данная функция принимает в качестве аргумента необязательный параметр callback - функцию, которая срабатывает при попытке совершить autoload.

ООП. Автозагрузка

При этом, если callback - функция не задана, то будет произведена попытка загрузить класс по-умолчанию. Данная попытка будет успешна в том случае, если будут соблюдены следующие правила:

- Каждый класс размещается в отдельном файле.
- Имя файла (+ расширение «.php») совпадает именем класса.
- Пространство имен (namespace) класса совпадает его расположением в каталогах.
- Может быть «базовый каталог», относительно которого указывается namespace.

К примеру, все файлы, хранятся в папке Application, а в корне проекта находится файл index.php. Если класс User описан в файле Application/User.php, то, вызвав в index.php функцию spl_autoload_register() и создав объект $user, программа отработает корректно:

```
$user = new \Application\User();
```

ООП. Автозагрузка

В случае, если какое-либо из условий не выполняется, то можно определить свое правило загрузки файлов:

```
const ROOT_PATH = __DIR__ . DIRECTORY_SEPARATOR . 'MyProject';

spl_autoload_register(function ($className) {
    $file = ROOT_PATH . DIRECTORY_SEPARATOR .
            preg_replace('/\\\\/', DIRECTORY_SEPARATOR, $className) . '.php';
    if (file_exists($file)) {
        require_once $file;
        return;
    }
    throw new Exception("{$className} and the file {$file} does not exist");
});
```

Теперь все файлы будут загружаться относительно базовой директории MyProject.

ООП. Магические методы

Магическими называются методы, с помощью которых определяется неявное поведение вашего объекта при различных манипуляциях с его экземпляром.
Магические методы начинаются с двойного подчеркивания.
Самым распространенным методом является конструктор:

```
class MyAwesomeClass
{
    public function __construct()
    {
        //some code here
    }
}
$object = new \MyProject\MyFolder\MyAwesomeClass();
```

ООП. Магические методы

Как видно, объект создается с помощью оператора new, а сам класс неявным образом вызывает метод __construct().
Деструктор также является магическим методом. Область его применения была рассмотрена выше.
Другими популярными магическими методами являются геттеры и сеттеры. В отличие от рассмотренных выше, магические аналоги работают неявно и могут привести к ошибкам.
Рассмотрим в качестве примера класс, каждый объект которого является записью в таблице базы данных.
Пусть существует таблица user, которая содержит поля id, name, email. Пусть есть класс User, который отображает конкретную запись user из данной таблицы. В качестве свойств будут выступать идентичные поля БД.

ООП. Магические методы

```php
class User
{
    private int $id;
    private string $name;
    private int $email;
}
```

Логичным было бы создать 3 геттера и 3 сеттера, которые бы возвращали и устанавливали бы значения для данных свойств. В данном примере это не проблематично и не затратно. Однако в случае, когда в проекте сотни таблиц, а в каких-то из них десятки полей, подобные классы могут содержать тысячи строк кода, который даже не реализует никакой бизнес-логики.
В качестве одного из решений можно рассмотреть магические геттеры и сеттеры. Это - унифицированные методы, которые вызываются, если у класса не определены соответствующие поля или методы.

ООП. Магические методы. Геттеры

```php
class User
{
    private int $id;
    private string $name;
    private int $email;
}
```

Логичным было бы создать 3 геттера и 3 сеттера, которые бы возвращали и устанавливали бы значения для данных свойств. В данном примере это не проблематично и не затратно. Однако в случае, когда в проекте сотни таблиц, а в каких-то из них десятки полей, подобные классы могут содержать тысячи строк кода, который даже не реализует никакой бизнес-логики.
В качестве одного из решений можно рассмотреть магические геттеры и сеттеры. Это - унифицированные методы, которые вызываются, если у класса не определены соответствующие поля или методы.

ООП. Магические методы. Геттеры

```php
class User
{
    private array $fields = ['id', 'name', 'email'];

    public function __get(string $name): string
    {
        if (in_array($name, $this->fields)) {
            return $this->getDatabaseField('id');
        }
        throw new Exception(
            'Поля ' . $name . ' не существует'
        );
    }
}

$user = new User();
echo $user->id;
```

Как видно из примера, у класса не заданы свойства.

При попытке обращения к свойству id будет неявно вызван метод __get(), которому качестве аргумента будет передано имя свойства (id).

С помощью не описанного метода getDatabaseField, логика реализации которого не важна в данном примере, возвращается значение поля id из БД.

Если же такого поля нет, будет вызвано исключение.

ООП. Магические методы. Сеттеры

```php
class User
{
    private array $fields = ['id', 'name', 'email'];
    private array $values = [];

    public function __set(string $name, mixed $value): void
    {
        if (in_array($name, $this->fields)) {
            $this->values[$name] = $value;
            return;
        }
        throw new Exception(
              'Поля ' . $name . ' не существует'
         );
    }
}
$user = new User();
$user->id = 1;
```

Логично, что имея магический метод читать свойства, есть метод для их записи. Такой метод называют сеттером.

Действует данный механизм аналогично магическим геттерам: если свойство класса не задано явно, произойдет вызов метода __set, который в качестве аргументов принимает название свойства и его значение.

ООП. Магические методы. Call

Как в случае с геттерами, существует и магический метод для обработки вызовов не определенных методов объекта.
Такой метод называется __call(string $name, array $arguments = []) и принимает в качестве аргументов название метода и список аргументов, которые передаются в метод, который необходимо вызвать.
Как видно из примера ниже, при вызове любого не описанного метода в классе, будет вызываться метод __call, который вернет строку, описывающую, какой метод вызывается.

```php
class User
{
    public function __call(string $name, array $arguments)
    {
        return 'Вызываем метод ' . $name . ' класса ' . __CLASS__;
    }
}

$user = new User();
echo $user->someMethod();
```

Для работы со статическими методами используется метод __callStatic(), по сути аналогичный методу __call()

ООП. Магические методы. toString

Метод __toString() позволяет определить логику работы приложения, при попытке привести объект к типу строка.
Применение данного метода актуально, к примеру, когда необходимо привести все свойства класса к JSON - строке.

При исполнении команды echo new User(); произойдет вызов конструктора, т.е. инициализация объекта, а после - попытка преобразования объекта в строку (приведение типа).

В этом случае интерпретатор попытается найти в классе реализацию метода __toString(), и если не найдет, то будет проброшено исключение.

```php
class User
{
    private int $id;
    private string $name;

    public function __toString(): string
    {
        return json_encode([
            'id' => $this->id,
            'name' => $this->name
        ]);
    }
}
echo new User();
```

ООП. Магические методы

Полный список магических методов представлен ниже. Не рассмотренные методы предлагается изучить самостоятельно:

- __construct(),
- __destruct(),
- __call(),
- __callStatic(),
- __get(),
- __set(),
- __isset(),
- __unset(),
- __sleep(),
- __wakeup(),
- __toString(),
- __invoke(),
- __set_state(),
- __clone(),
- __debugInfo().

ООП. Наследование

Наследование — это механизм, посредством которого один или несколько классов можно получить из некоторого базового класса.

Класс, который получается в результате наследования от другого, называется его подклассом. Эту связь обычно описывают с помощью терминов "родительский" и "дочерний".

Дочерний класс происходит от родительского и наследует его характеристики. Эти характеристики состоят из свойств и методов.

Обычно в дочернем классе к функциональности родительского класса (который также называют суперклассом) добавляются новые функциональные возможности. Поэтому говорят, что дочерний класс расширяет родительский.

ООП. Наследование

```
class User
{
    protected string $email;
    protected string $password;
}

class Profile extends User
{
    private string $firstName;
    private string $lastName;
}
```

Как видно, были созданы два класса, один из которых расширяет другой.

Класс User реализует логику работы с основными данными, такими как email, password, а Profile - с персональными.

Стоит отметить, что в базовом классе свойства обозначены, как protected, чтобы они были доступны в дочернем классе.

ООП. Наследование. Трейты

Особенность наследования в PHP состоит в том, что в языке не предусмотрено множественное наследование. Т.е. наследоваться одновременно от нескольких классов невозможно, это запрещено идиомой самого языка.
Для реализации подобного подхода применяется механизм трейтов.

```php
trait SoftDelete
{
    public function delete(): void
    {}
}

class User
{
}
```

```php
class Profile extends User
{
    use SoftDelete;
}

$profile = new Profile();
$profile->delete();
```

ООП. Наследование. Трейты

Как видно из примера, класс Profile наследуется от класса User и при этом использует функционал трейта SoftDelete.

При этом трейт является структурой, максимальной схожей с классом.

```php
trait SoftDelete
{
    public function delete(): void
    {}
}

class User
{
}
```

```php
class Profile extends User
{
    use SoftDelete;
}

$profile = new Profile();
$profile->delete();
```

ООП. Наследование

Наследование применяется по двум причинам: для реализации необходимой архитектуры, а также, для избежания дублирования кода.

Начинающие разработчики довольно часто применяют антипаттерн, который заключается в так называемом каскадном наследовании, когда несколько классов расширяются от предыдущих и выстраивается своеобразная вертикаль из классов.

Потенциальная проблема, которая может возникнуть, связана с потенциальными изменениями в одном из классов, находящимся вверху вертикальной цепочки при продолжающейся разработке.

Другая потенциальная проблема может заключаться в зацикливании наследования, т.е. класс A наследует класс B, а класс B наследует класс A.
В этом случае возникнет ошибка **Maximum function nesting level of '100' reached, aborting**

ООП. Абстракция

Абстракция — это придание объекту характеристик, которые четко определяют его концептуальные границы, отличая от всех других объектов.

Основная идея состоит в том, чтобы отделить способ использования составных объектов данных от деталей их реализации в виде более простых объектов, подобно тому, как функциональная абстракция разделяет способ использования функции и деталей ее реализации в терминах более примитивных функций.

Таким образом, данные обрабатываются функцией высокого уровня с помощью вызова функций низкого уровня.

В PHP абстракция реализуется с помощью абстрактных классов и интерфейсов.

ООП. Абстракция. Классы

Абстрактный класс - класс, который содержит хотя бы один абстрактный метод.

Абстрактный метод - метод, который имеет лишь сигнатуру и не имеет реализации.
Классы, являющиеся дочерними по отношению к абстрактному, должны реализовывать абстрактные методы.

В PHP наличие абстрактного метода в классе не является обязательным в терминах языка. Обязательным является ключевое слово abstract перед ключевым словом класс.

Метод объявляется абстрактным также с помощью ключевого слова abstract.

ООП. Абстракция. Классы

```php
abstract class Animal
{
    abstract public function say(): void;
}

class Cat extends Animal
{
    public function say(): void
    {
        echo 'Mew';
    }
}
```

Важной особенностью является то, что невозможно создать объект абстрактного класса.

Стоит отметить, что при объявлении абстрактных методов можно использовать любые модификаторы доступа.

Однако, использование модификатора private является достаточно нелогичным: использование абстрактного метода подразумевает под собой реализацию этого метода в классе-наследнике.

А приватный метод не может быть унаследован.

ООП. Интерфейсы

Интерфейс определяет абстрактный дизайн, которому должен соответствовать применяющий его класс. Интерфейс определяет методы без реализации. А класс затем применяет интерфейс и реализует эти методы. Применение интерфейса гарантирует, что класс имеет определенный функционал, описываемый интерфейсом.

Интерфейс определяется с помощью ключевого слова interface, за которым следует имя интерфейса и блок кода интерфейса в фигурных скобках.

```
interface Runnable
{
    public function run(): void;
}
```

ООП. Интерфейсы

Хорошим стилем именования интерфейсов является добавление суффикса Interface или префикса I к имени интерфейса.

```
interface IRunnable
{
    public function run(): void;
}

class Lion implements IRunnable
{
    public function run(): void
    {
        echo 'Run run run';
    }
}
```

Важные особенности интерфейсов:

1. Все методы в интерфейсе могут иметь только модификатор доступа public
2. Интерфейс может описывать только методы. Свойства в интерфейсе задавать невозможно
3. Методы в интерфейсе имеют исключительно сигнатуры, а реализация остается на откуп классам

ООП. Интерфейсы и абстрактные классы

Различие в применении абстрактного класса и интерфейса — очень тонкий вопрос.
Абстрактный класс скорее служит для объединения семейства классов.

Например, есть абстрактный класс Animal, и от него наследуются классы Cat, Lion, у которых есть общие методы (объявленные в абстрактном классе).

Но если появляется класс Fish, то в нем нет смысла реализовывать, например, метод say. Для таких классов лучше описывать интерфейсы.

Реализация абстракции возможна в комбинированном стиле - использовании как абстрактных классов, так и интерфейсов.

ООП. Интерфейсы и абстрактные классы

```php
interface INaming
{
    public function getName(): string;
}

interface IRunnable
{
    public function run(): void;
}

interface ISwimming
{
    public function swim(): void;
}

abstract class Animal implements INaming
{
    abstract public static function getAnimal(): self;
}
```

Реализованы три интерфейса:

- INaming, который описывает метод получения имени животного
- IRunnable, который описывает метод “бежать”
- ISwimming, который описывает метод “плыть”

Если INaming актуален для любого животного, то IRunnable и ISwimming - только для определенных семейств.

Реализованный абстрактный класс Animal реализует интерфейс INaming, а остальные классы - наследуются от Animal.

Стоит отметить, что абстрактный класс не обязан реализовывать методы интерфейса.

ООП. Интерфейсы и абстрактные классы

```php
class Lion extends Animal implements IRunnable
{
    public function run(): void
    {
        echo 'Run run run';
    }

    public function getName(): string
    {
        return 'Simba';
    }

    public static function getAnimal(): Lion
    {
        return new static();
    }
}
```

```php
class Fish extends Animal implements ISwimming
{
    public function swim(): void
    {
        echo 'Swim swim swim';
    }

    public function getName(): string
    {
        return 'Moony';
    }

    public static function getAnimal(): Fish
    {
        return new static();
    }
}
```

ООП. Полиморфизм

Язык PHP поддерживает полиморфизм в том смысле, что позволяет использовать вместо экземпляров родительского класса экземпляры подкласса.

Ввод в действие требуемого метода осуществляется на этапе прогона. Поддержка перегрузки методов, при которой ввод метода в действие осуществляется с учетом сигнатуры метода, отсутствует.

Дело в том, что в каждом классе может присутствовать только один метод с определенным именем. Но благодаря тому, что в языке PHP применяется слабая типизация и поддерживается переменное количество параметров, появляется возможность обойти это ограничение.

ООП. Полиморфизм

```php
abstract class Animal
{
    abstract public function getName(): string;
}

class Lion extends Animal
{
    public function getName(): string
    {
        return 'Simba';
    }
}

class Fish extends Animal
{
    public function getName(): string
    {
        return 'Moony';
    }
}
```

```php
$animals = [
    new Lion(),
    new Fish()
];

foreach ($animals as $animal) {
    echo $animal->getName();
}
```

В данном случае полиморфизм проявляется в переопределении абстрактного метода getName().

ООП. Внедрение зависимостей (DI)

Внедрение зависимостей — это стиль настройки объекта, при котором поля объекта задаются внешней сущностью. Другими словами, объекты настраиваются внешними объектами.

DI — это альтернатива самонастройке объектов. Это может выглядеть несколько абстрактно, так что посмотрим пример:

ООП. Внедрение зависимостей (DI)

```php
class HouseBuilder
{
    public function __construct(
        private float $square,
        private float $wallsLength,
        private float $wallsHeight,
    ) {}

    private function buildFoundation(): void
    {
        echo 'Build the foundation with square '
            . $this->square;
    }

    private function buildWalls(): void
    {
        echo 'Build the walls ' .
            $this->wallsLength . 'x' .
            $this->wallsHeight;
    }
```

```php
    public function build(): void
    {
        $this->buildFoundation();
        $this->buildWalls();
    }
}
```

В данном примере класс HouseBuilder владеет информацией о строительстве всего дома и достаточно подробно: какова площадь фундамента и каковы длина и ширина стен.

Данный подход можно сравнить с микроменеджментом в компании, где руководитель вникает в каждую мелочь в работе подразделений.
Но когда компания становится больше, руководителю все сложнее заниматься всеми нюансами и он делегирует.
Аналогично с со строительством дома: главный класс не должен заниматься всем сам и владеть информацией обо всех процессах.
Применим подход Dependency Injection.

ООП. Внедрение зависимостей (DI)

```php
interface IBuilder
{
    public function build(): void;
}

class WallsBuilder implements IBuilder
{
    public function __construct(
        private float $wallsLength,
        private float $wallsHeight
    ) {}

    public function build(): void
    {
        echo 'Build the walls ' .
             $this->wallsLength . 'x' .
             $this->wallsHeight;
    }
}
```

```php
class FoundationBuilder implements IBuilder
{
    public function __construct(
        private float $square
    ) {}

    public function build(): void
    {
        echo 'Build the foundation with square ' .
             $this->square;
    }
}

class HouseBuilder implements IBuilder
{
    public function __construct(
        private FoundationBuilder $foundationBuilder,
        private WallsBuilder $wallsBuilder
    ) {}

    public function build(): void
    {
        $this->foundationBuilder->build();
        $this->wallsBuilder->build();
    }
}
```

ООП. Внедрение зависимостей (DI)

Как видно, вместо того, чтобы класс HouseBuilder владел всей информацией, он делегирует каждому дочернему классу свою зону ответственности, а сам использует лишь метод build этих классов.

Важно, что при создании объекта HouseBuilder необходимо предварительно создать объекты остальных классов. В случае, если таких классов много, это влечет за собой много затрат на создание всех этих объектов.

Если учесть, что каждый зависимый объект также может содержать зависимости, а они - другие зависимости, то верным решением является автоматизировать функцию внедрения зависимостей.

Стоит отметить, что современные популярные фреймворки, такие как Laravel, Yii2, Symfony, реализуют DI “из коробки”.

ООП. Внедрение зависимостей (DI)

Реализуется механизм внедрения зависимостей с помощью Reflection API.

```php
function di(string $class): object
{
    $classReflector = new \ReflectionClass($class);
    $constructReflector = $classReflector->getConstructor();

    if (empty($constructReflector)) {
        return new $class;
    }

    $constructArguments = $constructReflector->getParameters();

    if (empty($constructArguments)) {
        return new $class;
    }

    $args = [];
    foreach ($constructArguments as $argument) {
        $argumentType = $argument->getType()->getName();
        $args[$argument->getName()] = $this->get($argumentType);
    }

    return new $class(...$args);
}
```

ООП. Наследование и композиция

Недостатки наследования были рассмотрены выше. Нивелировать их призван паттерн проектирования “композиция”.

```
class HouseBuilder
{
    public function __construct(
        private float $square,
        private float $wallsLength,
        private float $wallsHeight,
    ) {}

    private function buildFoundation(): void
    {
        echo 'Build the foundation with square ' .
             $this->square;
    }

    private function buildWalls(): void
    {
        echo 'Build the walls ' .
             $this->wallsLength . 'x' .
             $this->wallsHeight;
    }

    public function build(): void
    {
        $this->buildFoundation();
        $this->buildWalls();
    }
}
```

В ситуации, когда необходимо расширить функционал, например, создать класс CottageBuilder, и добавить функцию прокладки трубопровода, будет достаточно сложно реализовать это в текущей архитектуре.

ООП. Композиция

```php
class CottageBuilder extends HouseBuilder
{
    public function buildPipeline(): void
    {
        echo 'Building Pipeline';
    }
}
$cottageBuilder = new CottageBuilder();
$cottageBuilder->build();
$cottageBuilder->buildPipeline();
```

```php
class CottageBuilder implements IBuilder
{
    public function __construct(
        private HouseBuilder $houseBuilder
    ) {}

    private function buildPipeline(): void
    {
        echo 'Build Pipeline';
    }

    public function build(): void
    {
        $this->houseBuilder->getFoundationBuilder()->build();
        $this->buildPipeline();
        $this->houseBuilder->getWallsBuilder()->build();
    }
}
```

В данном примере прокладка трубопровода возможна либо до заливки фундамента, либо после строительства стен. Поэтому, используем другой способ, который был описан в примере DI.

Важно: в класс HouseBuilder добавлены геттеры `getFoundationBuilder()` и `getWallsBuilder()`