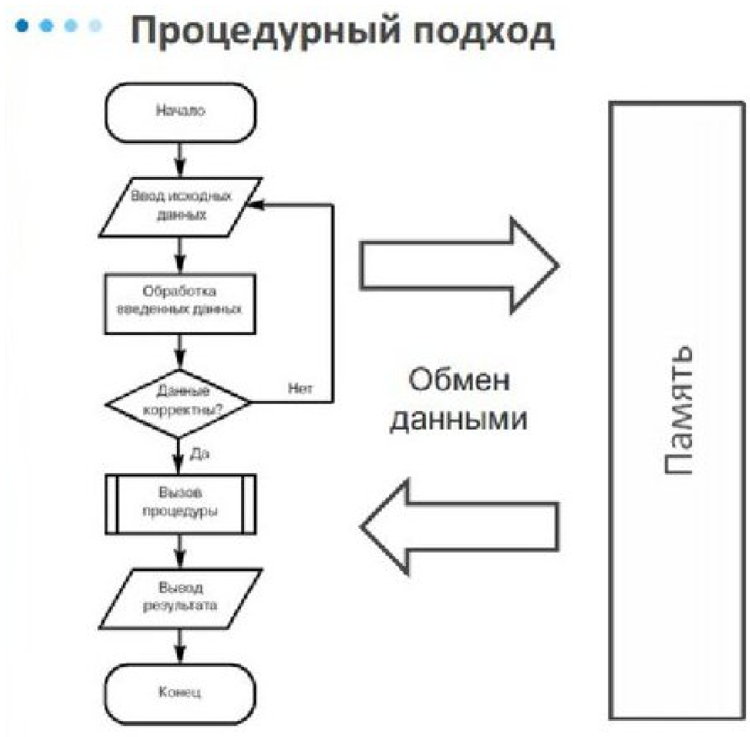
5. Реализация ООП в PHP

Язык программирования PHP

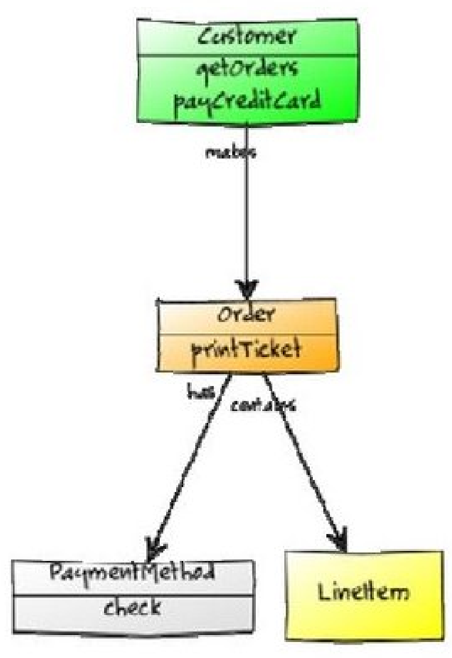
- Императивный процедурный
- Функциональные возможности
- **Поддержка ООП на классах (а-ля Java)**

Исторически PHP возник как процедурный язык (примитивы языка не являются объектами), в который постепенно добавлялись элементы ООП.
В каждой версии языка возможности ООП расширяются.

Две парадигмы программирования



Объектно-ориентированный



Программа – алгоритм последовательного вызова проедур изменения данных в памяти.

Программа – взаимодействие объектов, компонентов, отсылка и обработка событий.

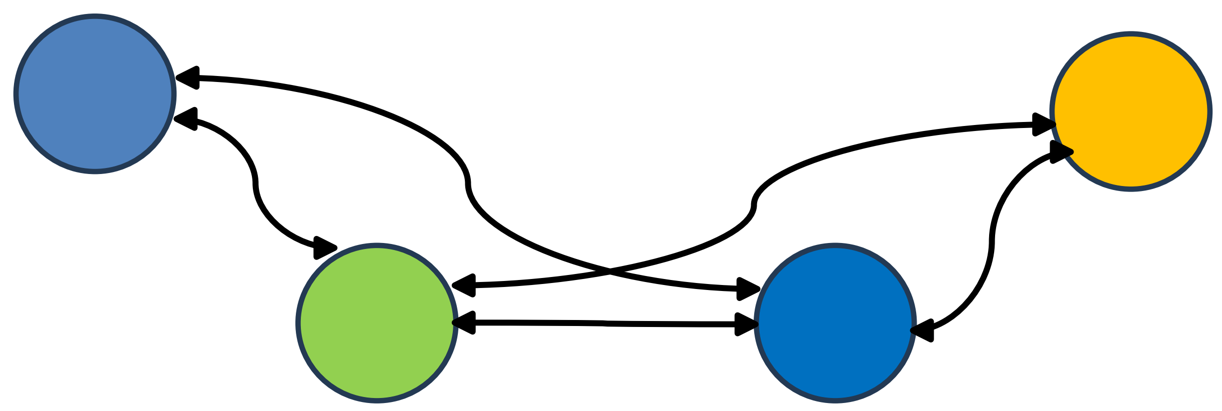
Программа = Алгоритмы +
Структуры данных

(Н. Вирт)

Приложение строится из «кирпичей» - объектов

Суть ООП

- Людям свойственно воспринимать окружающий мир как множество взаимодействующих между собой объектов, поддающихся определенной классификации.
- ООП - попытка связать поведение сущности с её данными и спроецировать объекты реального мира и бизнес-процессов в программный код.



Объекты и классы

Объект имеет имя и определяется:

- **Атрибутами/свойствами** (*размер, цвет, вес, …*)
- **Поведениями/действиями** (*бежать, говорить, ...*)

House
- address - numberFloors - numberWindows
+ build() + destroy() + repair()

Human
- age - height - weight
+ speak() + walk() + sleep()

TV
- manufacturer - model - size
+ on() + switchChannels() + off()

Объект — обладающий именем и физически находящийся в памяти компьютера набор **данных** и **методов**, имеющих доступ к ним.

- **Структура данных**

- **Операции (подпрограммы)**

Объект = Данные + Методы

Объекты и события

Объекты могут:

- Взаимодействовать друг с другом посредством **сообщений**.
- Реагировать на определенные **события** (обрабатывать их), возникающие вследствие действий пользователя или других объектов.

Хорошо подходит для графического интерфейса (обработчики событий нажатия на кнопку и т.п.)

Объект

Свойства

Методы

Обработчики событий

ООП на классах

Класс – шаблон , на основе которого создаются объекты-экземпляры класса (с одним и тем же набором свойств и методов).
Реальный объект должен иметь конкретные значения всех полей.

Класс – пользовательский тип, который задает поведение будущих "переменных" - объектов.

Готовые классы в PHP

- DateTime
- PDO
- DirectoryIterator
- ZipArchive
- ArrayObject
- ...

```
$date = new DateTime();

// Выведет текущую дату и время
// в формате ГГГГ-ММ-ДД ЧЧ:ММ:СС
echo $date->format('Y-m-d H:i:s');
```

Есть Standard PHP Library (SPL) - набор интерфейсов и классов, которые решают общие задачи (структуры данных для АТД, классы для работы с файловой системой, …)

Описание класса и создание объекта в PHP

```php
<?php
class Student
{
    public string $name;
    public string $lastName;

    function showFullName()
    {
        echo "Полное имя: ".$this->name." ".$this->lastName.PHP_EOL;
    }
}

// ========== Create objects ============================

$student1 = new Student;

$student1->name = "Сергей";
$student1->lastName = "Иванов";
$student1->showFullName();
```

Конструктор

Метод `__construct()`.

Основное назначение – инициализация атрибутов объекта.

```
class Student
{
    public string $name;
    public string $lastName;

    public function __construct(string $name, string $lastName)
    {
        $this->name = $name;
        $this->lastName = $lastName;
    }
}

$student1 = new Student("Сергей", "Иванов");
```

Конструктор

1. Создается объект (технически это структура из языка C).
2. Вызывается конструктор, в который передается этот объект под именем `$this`.
3. Свойствам объекта `$this` присваиваются значения.
4. Объект возвращается наружу, срабатывает оператор присваивания.

```
class Student
{
    public string $name;
    public string $lastName;

    public function __construct(string $name, string $lastName)
    {
        $this->name = $name;
        $this->lastName = $lastName;
    }
}

$student1 = new Student("Сергей", "Иванов");
```

Конструктор

Начиная с PHP 8, свойства можно объявлять в параметрах конструктора (Constructor Property Promotion). Инициализация пройдет автоматически.

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

$student1 = new Student("Сергей", "Иванов");
$student2 = new Student(name: "Иван", lastName: "Сергеев");
```

Магические методы

Названия заранее определены, позволяют влиять на поведение объекта на разных этапах жизненного цикла.

- `__construct()` Вызывается в момент созданя объекта
- `__destruct()` Вызывается в момент уничтожения объекта
- `__call()` Вызывается при попытке вызвать несуществующий метод
- `__toString()` Вызывается при интерполяции объекта в строку
- `...`

Магические методы

```php
<?php
class User {
    private $name;

    public function __construct($name) {
        $this->name = $name;
    }

    // Магический метод __toString()
    public function __toString() {
        return "Пользователь: {$this->name}";
    }

    public function __invoke($arg) {
        echo "Привет, $arg!";
    }
}

// Создаем экземпляр класса User
$user = new User("Иван");

// Выводим объект как строку
echo $user . PHP_EOL;  // Выведет: Пользователь: Иван

$user("Иван"); // Выведет: Привет, Иван
```

Именование методов

Принцип Command Query Segregation (CQS)

Command

```
$advert->edit($content)
$advert->move($categoryId)

$advert->addPhoto($file)
$advert->removePhoto($id)

$advert->assignTag($tagId)
$advert->revokeTag($tagId)

$advert->sendToModeration($date)
$advert->moderate($date)
$advert->reject($reason)
$advert->close()
```

Query

```
$advert->getId()
$advert->getDate()
$advert->getUserId()
$advert->getCategoryId()
$advert->getContent()
$advert->getPhotos()
$advert->getTags()
$advert->getPublishDate()
$advert->getExpireDate()
$advert->getRejectReason()

$advert->isDraft()
$advert->isOnModeration()
$advert->isActive()
$advert->isClosed()
$advert->isPublishedFor($date)

$advert->hasPhotos()
$advert->hasTag($tagId)
$advert->canBePublished()
```

Присваивание объектов и передача их в функции

```
$student1 = new Student; $student2 = $student1;
```

- При создании объекта в переменную записывается указатель (pointer) на него. Это идентификатор (номер) объекта, находящегося в памяти.
- При присваивании или передаче объекта в функцию происходит копирование идентификатора (сам объект не меняется и не дублируется).

Это похоже на присваивание и передачу параметров по ссылке для обычных переменных.

Создать копию объекта можно оператором `clone`

```
$second = clone $first;
```

Сравнение объектов

- Объекты разных типов никогда не равны
- Нестрогое сравнение (==). Два объекта равны, если они имеют одинаковые свойства и их значения совпадают.
- Строгое сравнение (===). Объекты строго равны, только если это один и тот же объект.

```
$p1 = new Point(3, 9);
$p2 = new Point(3, 9);

$p1 == $p2;  // true
$p1 === $p2; // false

$p3 = $p1;
$p3 === $p1; // true
```

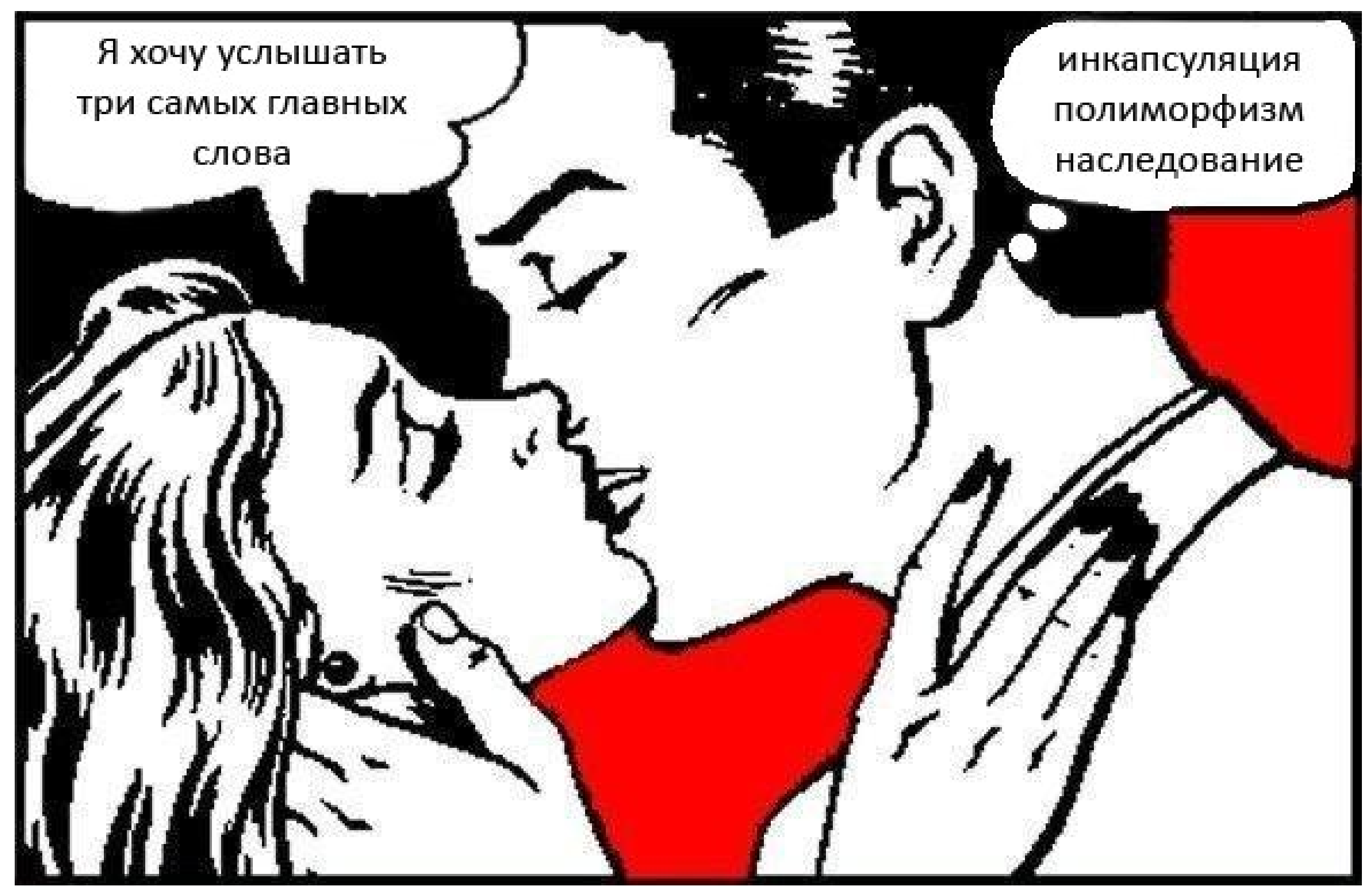
Причины создания классов

- Моделирование объектов реального мира
- Моделирование абстрактных объектов
- Хранение конфигурации для выполнения действий
- Сокрытие глобальных данных
- Упаковка родственных операций

Виды объектов:

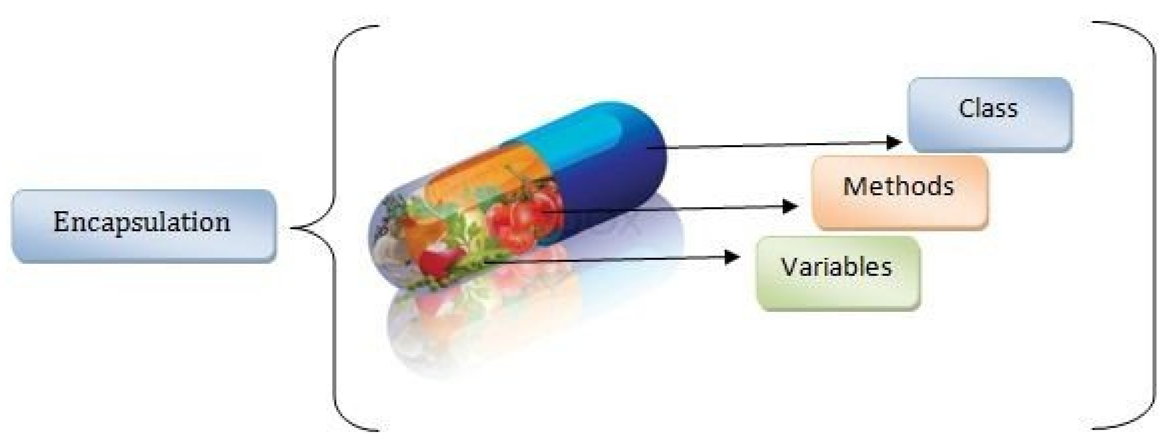
- Объекты-сущности
- Объекты-значения

Основные принципы ООП



Инкапсуляция и сокрытие данных

Инкапсуляция - упаковка данных и функций в один компонент.



Сокрытие данных – пользователь класса может работать только с его интерфейсной частью и не имеет доступа к реализации функциональности класса.

Инкапсуляция (программирование)

Материал из Википедии — свободной энциклопедии [править | править код]

Текущая версия страницы пока не проверялась опытными участниками и может значительно отличаться от версии, проверенной 21 сентября 2016; проверки требуют 44 правки.

У этого термина существуют и другие значения, см. Инкапсуляция.

Инкапсуляция (англ. *encapsulation*, от лат. *in capsula*) — в информатике упаковка данных и функций в единый компонент.

Любая программная сущность, обладающая нетривиальным состоянием, должна быть превращена в замкнутую систему, которую можно только перевести из одного корректного состояния в другое.

Принцип сокрытия данных

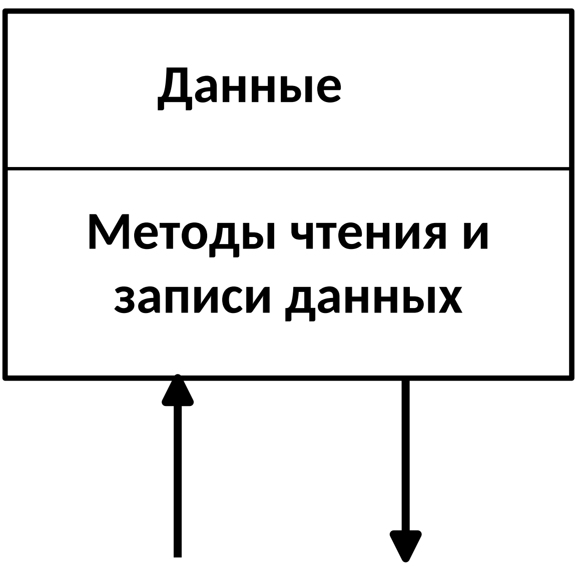
Сокрытие реализации класса и отделение его внутреннего представления от внешнего интерфейса.

Интерфейс **Реализация**

Принцип инкапсуляции и сокрытия данных

Доступ к данным класса возможен только посредством методов этого класса.

Свойство объекта (property) – совокупность атрибута (поля) объекта и методов его чтения/записи.



Принцип инкапсуляции и сокрытия данных

- Разработчик в методах контролирует данные, передаваемые снаружи, и не позволяет записать недопустимые значения в поля объекта.
- Разработчик класса определяет, что можно делать с объектом, а что нельзя.
- Можно менять внутреннюю логику работы класса, не меняя публичные методы, и весь остальной код, который их использует.
- Упрощается понимание кода: чтобы понять, как использовать класс, достаточно прочитать названия публичных методов.
- Отдельные части программы (объекты) можно разрабатывать и тестировать независимо друг от друга.

Механизм сокрытия данных

Области видимости свойств и методов класса в PHP:

- **public** — доступны отовсюду
- **private** — доступны только из методов данного класса

Методы-аксессоры

- `__get($name)` вызывается при попытке доступа к несуществующему или недоступному свойству `$name` объекта.
- `__set($name, $value)` вызывается при попытке записать в несуществующее или недоступное свойство `$name` объекта значение `$value`.

Методы-аксессоры

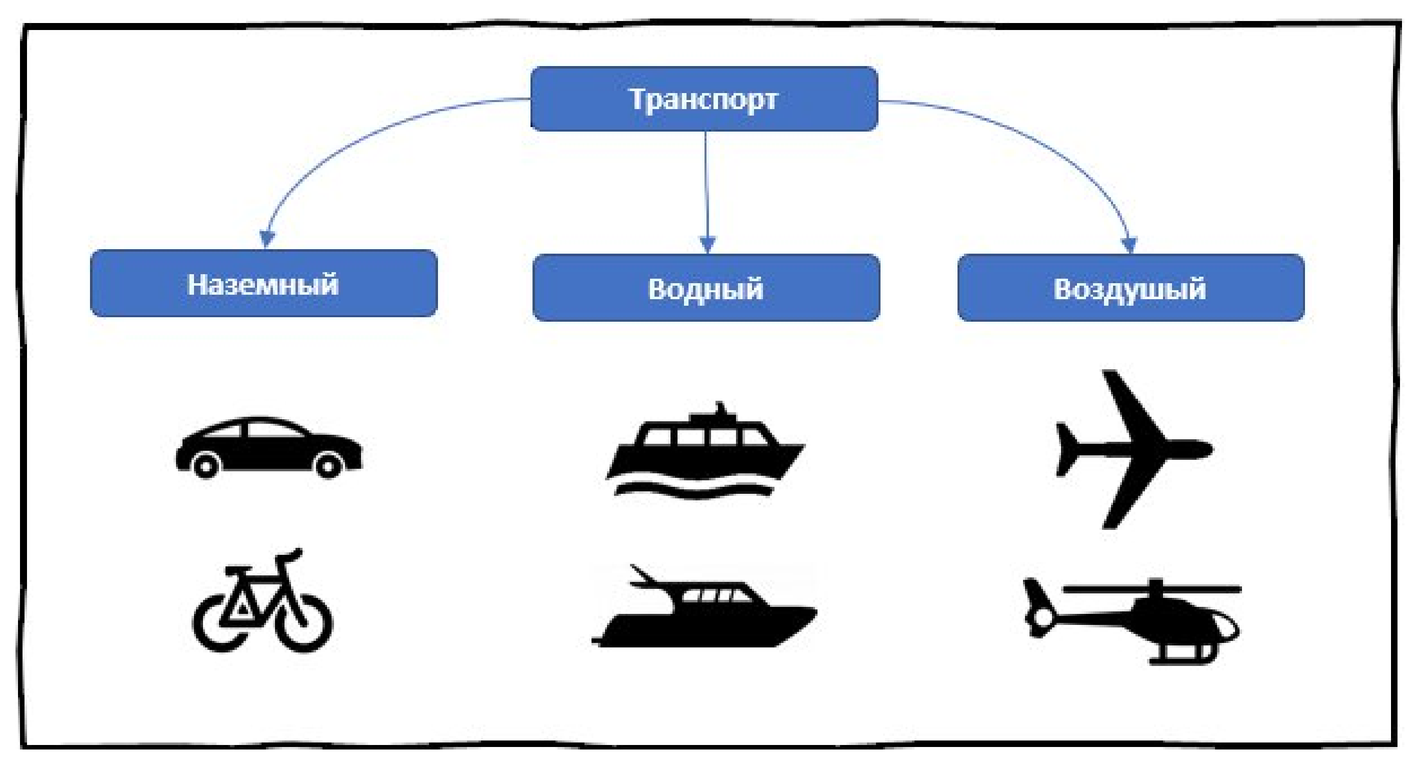
```
class User {
    private $data = [];

    public function __get($name) {
        return $this->data[$name] ?? null;
    }

    public function __set($name, $value) {
        $this->data[$name] = $value;
    }
}

$user = new User();
$user->name = 'John'; // Вызывает __set('name', 'John')
echo $user->name;     // Вызывает __get('name') -> Выведет: John
```

Объект с динамической структурой данных

Наследование

Наследование

Наследование (иерархия) - возможность порождать один класс от другого с сохранением по наследству атрибутов и поведений от родительских классов, добавляя, при необходимости, новые свойства и методы.



Отношение **“является”**

Принцип наследования/расширения

- Оператор `extends` - расширение класса
- Область видимости **protected** — поле или метод доступны в классе, котором они определены и во всех его дочерних классах.

В потомке можно:

- Перекрыть метод родителя.
- Вызвать одноименный метод родителя через `parent::`*`method()`*

Абстрактные классы

Классы могут наследоваться от **абстрактного класса**, который не может иметь собственных экземпляров.

Абстрактный класс - не базовое понятие ООП, во многих языках (Ruby, JavaScript, Python) такого понятия нет.

Статические свойства и методы

- Определяются в контексте класса, а не объекта
- Статические методы и свойства доступны из любой точки программы
- Значения статических свойств одинаковы для всех экземпляров класса
- Не нужно создавать экземпляры класса только ради вызова простой функции

Статические свойства и методы

- Доступ через оператор `::` и ключевые слова `self` или `static`
- `$this` - позднее связывание, `self` - раннее связывание, `static` - позднее связывание

Статические свойства и методы

```php
<?php
class Counter {
    public static $count = 0; // Статическое свойство

    public function increment() {
        self::$count++; // Увеличиваем счётчик
    }

    public static function getCount() {
        return self::$count; // Возвращаем текущее значение счётчика
    }
}

Counter::$count = 100; // Устанавливаем начальное значение счётчика

$counter1 = new Counter();
$counter1->increment(); // Увеличили счётчик до 101

$counter2 = new Counter();
$counter2->increment(); // Увеличили счётчик до 102

echo Counter::getCount(); // Выведем 102
```

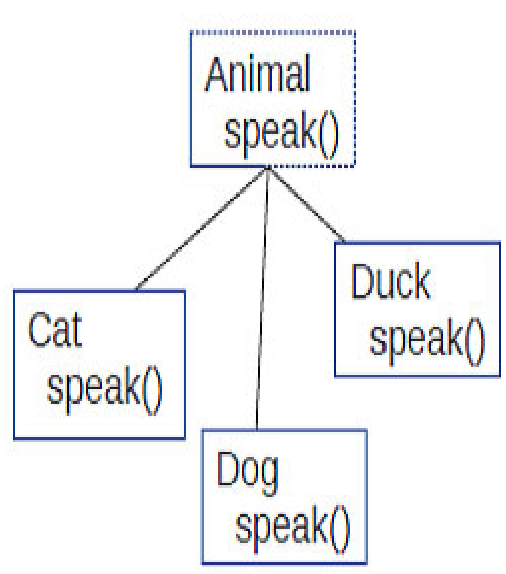
Статические свойства и методы

- Статические поля аналогичны глобальным переменным:
  - их изменение влияет на весь код,
  - они добавляют побочные эффекты в использующие их функции.
- Код на статических методах, по сути, не является объектно-ориентированным. При росте приложения изменять его станет сложно.
- Статические методы подходят для простых функций, не использующих поля класса, результат которых зависит только от переданных аргументов.

Полиморфизм

Полиморфизм

Функции с одним и тем же именем соответствует разный программный код в зависимости от того, объект какого класса используется при вызове этой функции.



Полиморфизм = "многообразие форм"

Полиморфизм - свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Принцип полиморфизма (позднее связывание)

Объект подкласса (потомка) может использоваться всюду, где используется объект суперкласса (предка).

- При добавлении к иерархии классов нового подкласса не нужно менять написанный код.
- «Позднее связывание» позволяет определять версию полиморфного (виртуального) метода во время выполнения программы.

Исследование классов (рефлексия)

Исследование класса

Рефлексия (Reflection) — это возможность программы **изучать саму себя** во время выполнения.

С помощью `ReflectionClass` вы можете получить полную информацию о классе: его свойства, методы, параметры конструктора, атрибуты, модификаторы доступа и многое другое — **без создания экземпляра класса**.

1. Простая аналогия 🔍

Представьте, что у вас есть закрытая коробка (класс).

Без рефлексии	С рефлексией
Вы можете только пользоваться предметом внутри	Вы можете открыть коробку и изучить, что внутри
Не знаете, какие есть методы	Видите все методы и свойства
Не знаете типы параметров	Видите типы аргументов конструктора

ReflectionClass — это "рентген" для ваших классов.

Исследование класса

```php
<?php
$date = new DateTime();

// Выведет текущую дату и время
// в формате ГГГГ-ММ-ДД ЧЧ:ММ:СС
echo $date->format('Y-m-d H:i:s');

echo gettype($date) . PHP_EOL; // object
echo get_class($date) . PHP_EOL; // DateTime
echo $date::class . PHP_EOL;      // DateTime

$reflection = new ReflectionClass('DateTime');

// Получаем все свойства класса
$properties = $reflection->getProperties();
foreach ($properties as $property) {
    echo "Имя свойства: " . $property->getName() . "\n";
}

// Получаем все методы класса
$methods = $reflection->getMethods();
foreach ($methods as $method) {
    echo "Имя метода: " . $method->getName() . "\n";
}
```

Исследование класса

Метод	Что делает
`getName()`	Возвращает имя класса
`getMethods()`	Все методы класса
`getProperties()`	Все свойства класса
`getConstructor()`	Конструктор класса
`hasMethod($name)`	Есть ли метод
`hasProperty($name)`	Есть ли свойство
`isInstantiable()`	Можно ли создать объект
`getAttributes()`	Получение PHP-атрибутов (PHP 8+)
`newInstanceArgs($args)`	Создание объекта с аргументами

Где используется во фреймворках

Фреймворк	Использование рефлексии
Laravel	DI Container, Routing, Middleware, Validation
Symfony	DI Container, Routing, Serializer, Validator
PHP-DI	Autowiring зависимостей
Doctrine	ORM маппинг сущностей
PHPUnit	Тестирование private методов

Пример: Сериализация/Десериализация

php

```php
class Serializer {
    public function toArray($object): array {
        $reflector = new ReflectionClass($object);
        $data = [];

        // Получаем все свойства (включая private)
        foreach ($reflector->getProperties() as $property) {
            $property->setAccessible(true); // Делаем доступными
            $data[$property->getName()] = $property->getValue($object);
        }

        return $data;
    }
}

$user = new User(1, 'John');
$serializer = new Serializer();
print_r($serializer->toArray($user));
```

Где используется: Symfony Serializer, Laravel API Resources.

Интерфейсы

Интерфейсы

В интерфейсе декларируется функциональность (обычно какое-то умение), которую должен реализовать каждый класс, реализующий этот интерфейс.

Что здесь?

Любое устройство, реализующее интерфейс «Вилка для электрической розетки»

Интерфейсы

Аналог абстрактного класса, содержащего только абстрактные методы.

Интерфейсы — это облегченные классы, которые диктуют нам условия.

- Интерфейсы можно наследовать друг от друга.
- Класс может реализовывать несколько интерфейсов (наследоваться от нескольких классов нельзя).
- Интерфейсы можно размещать в пространствах имен.

- Класс = тип данных + шаблон для создания экземпляров
- Интерфейс = тип данных

Интерфейсы и PSR-стандарты

- PSR-3 (LoggerInterface) - определяет стандартный интерфейс для логгеров.
- PSR-7 (HTTP Message Interface) - интерфейсы для HTTP-сообщений.
- PSR-11 (ContainerInterface) - интерфейс для контейнеров зависимостей.

Интерфейс `Psr\Log\LoggerInterface` содержит **9 методов**:

Метод	Уровень (Level)	Когда использовать
`emergency()`	`emergency`	Система неработоспособна (сайт упал, БД недоступна)
`alert()`	`alert`	Требует немедленного действия (потеря данных, безопасность)
`critical()`	`critical`	Критические ошибки (исключение в ядре, непредвиденная ситуация)
`error()`	`error`	Ошибки runtime (исключение не поймано, функция вернула false)
`warning()`	`warning`	Предупреждения (некритичные ошибки, устаревшие API)
`notice()`	`notice`	Нормальные, но важные события (новый пользователь, заказ)
`info()`	`info`	Информационные сообщения (запрос выполнен, кэш очищен)
`debug()`	`debug`	Отладочная информация (значения переменных, SQL-запросы)
`log()`	**Любой**	Универсальный метод, куда передаётся уровень вручную

php

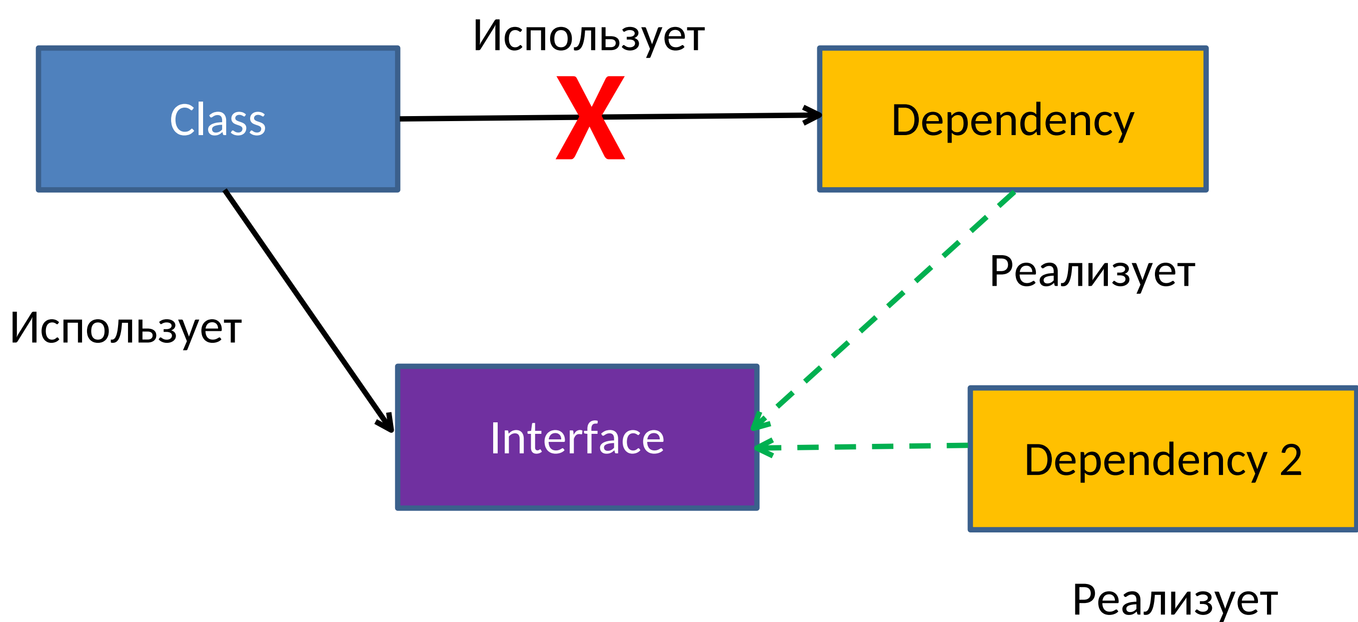
```php
<?php
use Monolog\Logger;
use Monolog\Handler\StreamHandler;
use Psr\Log\LogLevel;

// 1. Создаём логгер
$logger = new Logger('my_app');
$logger->pushHandler(new StreamHandler('app.log', Logger::DEBUG));

// 2. Используем методы уровней
$logger->info('Приложение запущено');
$logger->warning('Заканчивается место на диске');
$logger->error('Ошибка подключения к БД', ['db_host' => 'localhost']);

// 3. Используем универсальный метод log()
$logger->log(LogLevel::NOTICE, 'Пользователь {user} выполнил действие {action}', [
    'user' => 'admin',
    'action' => 'delete_post'
]);

// 4. В файле app.log появится:
// [2023-10-27T10:00:00+00:00] my_app.INFO: Приложение запущено []
// [2023-10-27T10:00:00+00:00] my_app.ERROR: Ошибка подключения к БД {"db_host":"localhost
```

6. Зачем это нужно? (Преимущества PSR-3)

1. **Совместимость:** Вы пишете код один раз, а использовать можете Monolog, Symfony Logger, Laravel Logger или любой другой.
2. **Гибкость:** В разработке можно логировать всё (`debug`), а на продакшене только ошибки (`error`), меняя только конфиг логгера.
3. **Стандартизация:** Любой PHP-разработчик знает, что делает `$logger->error()`.

Интерфейсы и Dependency Inversion Principle

Основная цель DIP — сделать код более гибким и переиспользуемым за счет использования абстракций (интерфейсов или базовых классов) вместо конкретных реализаций.



Интерфейсы в PHP-фреймворках

- Внедрение зависимостей (Dependency Injection)
- Создания абстракций над конкретными реализациями
- Обеспечения расширяемости и заменяемости компонентов
- Упрощения тестирования через мокирование зависимостей

Примеси (трейты)

Примеси

Трейт — общий программный код, который можно добавлять к любым классам PHP.

- Включаются в класс оператором `use` - аналог `include`, действие которого распространяется только на конкретный класс.
- Трейты не изменяют тип класса, в который они включаются.
- Выглядит и устроен как (абстрактный) класс.
- Трейт не может реализовывать интерфейс.
- Внутри класса к методам трейта можно обращаться только через `$this`.

Примеси - зачем еще одно понятие?

Трейты - механизм переиспользования общего кода в разных классах, альтернативный наследованию.

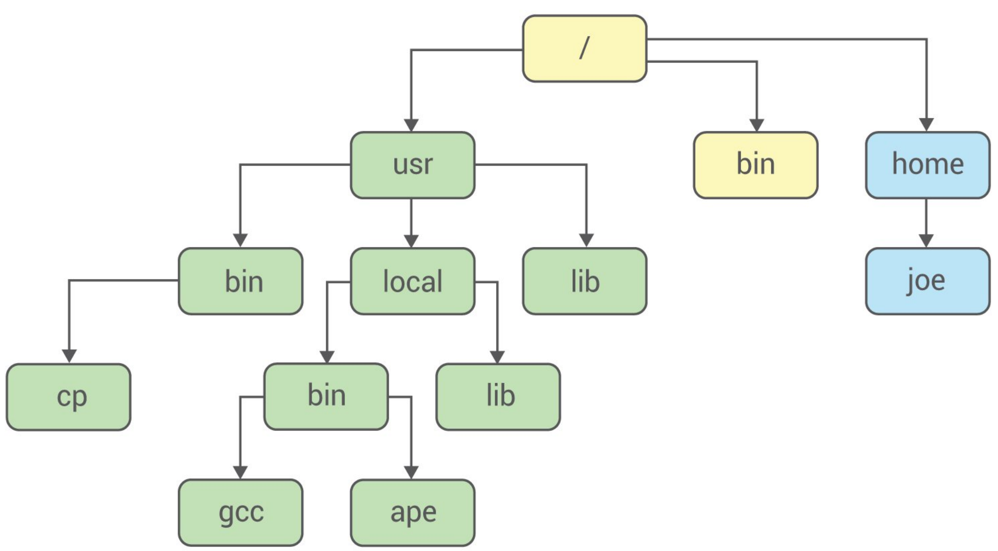
Трейты в отличие от наследования, не фиксируют структуру классов.

Пространства имен и автозагрузка классов

Пространства имен для классов

Классы, функции и константы можно изолировать по разным пространствам имен, образующим иерархию.

- Предотвращение конфликта имен
- Создание псевдонимов для длинных имен.



- **Соответствие между именами классов и путями к их файлам. Автозагрузка классов при обращении к ним.**

Автозагрузка классов

- В PHP с помощью функции `spl_autoload_register` можно регистрировать функции-автозагрузчики.
- При обращении к несуществующему классу PHP по очереди вызывает зарегистрированные автозагрузчики, передавая им имя класса.
- Автозагрузчик должен найти файл с нужным классом и загрузить его.
- Если ни один автозагрузчик не подключит файл с классом, то будет выведена ошибка об обращении к несуществующему классу.

Автозагрузка классов

Требования PSR-4 и PSR-12 указывают, где размещать и как называть файлы с классами, интерфейсами, трейтами, чтобы они автоматически загружались при обращении к ним.

- Ровно один класс на файл. В файле нет других инструкций, кроме `namespace` и описания класса.
- Название класса в StudlyCase.
- Файл называется в точности, как класс, с учетом регистра (+расширение php).

Исключения

Исключения

Исключения (exceptions) - способ обработки ошибок. Ошибки не должны обрабатываться кодом, где они возникли.

- Главная цель механизма исключений - передать ошибку из того места, где она возникла, в место, где её можно обработать, минуя все промежуточные уровни.
- Код, в котором произошла ошибка, выбрасывает (throw) исключение, а код, в котором она обрабатывается - ловит (catch) исключение.

Базовая конструкция исключений

```php
<?php
echo "Начало программы\n";

try {
    // Код, в котором перехватываются исключения
    echo "Все, что имеет начало...\n";

    // Генерируем ("выбрасываем") исключение
    throw new Exception("Случилось страшное!", 100);

    echo "...имеет и конец";
} catch (Exception $exp) {
    // Код обработчика
    echo " Исключение: {$exp->getMessage()}, код: {$exp->getCode()}\n";
}

echo "Конец программы\n";
```

```
Начало программы
Все, что имеет начало...
 Исключение: Случилось страшное!, код: 100
Конец программы
```

Раскрутка стека вызовов

```php
<?php
echo "Начало программы\n";

try {
    echo "Начало try-блока\n";
    outer();
    echo "Конец try-блока\n";
} catch (Exception $e) {
    echo " Исключение: {$e->getMessage()}\n";
}

echo "Конец программы\n";

function outer() {
    echo "Вошли в функцию ".__METHOD__."\n";
    inner();
    echo "Вышли из функции ".__METHOD__."\n";
}

function inner() {
    echo "Вошли в функцию ".__METHOD__."\n";
    throw new Exception("Hello!");
    echo "Вышли из функции ".__METHOD__."\n";
}
```

```
Начало программы
Начало try-блока
Вошли в функцию outer
Вошли в функцию inner
 Исключение: Hello!
Конец программы
```

Интерфейс класса Exception

Исключение - объект класса Exception или его наследника. Этот объект содержит внутри себя сообщение, переданное в конструктор, трассировку стека и другие полезные данные.

```
<?php

Exception implements Throwable {
    protected string $message ;
    protected int $code ;
    protected string $file ;
    protected int $line ;

    public __construct ([ string $message = "" [, int $code = 0 [, Throwable
$previous = NULL ]]] )
    final public getMessage ( void ) : string
    final public getPrevious ( void ) : Throwable
    final public getCode ( void ) : mixed
    final public getFile ( void ) : string
    final public getLine ( void ) : int
    final public getTrace ( void ) : array
    final public getTraceAsString ( void ) : string
    public __toString ( void ) : string
}
```

Отношения между классами

Взаимодействие объектов

Запросить текущее состояние
(вызов метода-запроса)

- **Ответить на запрос**
  (`return` …)
- **Не ответить на запрос**
  (`exception` …)
- **Делегировать запрос**
  (вызов метода-запроса)

Взаимодействие объектов

Указать выполнить действие
(вызов метода-команды)

- **Выполнить действие**
- **Отказаться, назвав причину**
  (exception …)
- **Прервать выполнение**
  (exception …)
- **Записать задачу в ежедневник**
  (поставить в очередь)
- **Сообщить о выполнении**
  (событие)
- **Делегировать приказ**
  (вызов метода-команды)

Отношения между классами

- **Наследование**

Кошка ***является*** *животным*

- **Ассоциация**

В свойстве класса содержится ссылка на экземпляр(ы) другого класса

  - **Композиция**

    Квартира ***имеет*** *комнату.*

  - **Агрегация**

    Автомобиль ***имеет*** *колесо.*

Агрегация и композиция

Отношение **агрегации**

Отношение **композиции**

Композиция

Объект A управляет временем жизни объекта B

```php
class Engine
{
    private $power;

    public function __construct($power)
    {
        $this->power = $power;
    }
}

class Car
{
    private $model;
    private $engine;

    public function __construct($model, $power)
    {
        $this->model = $model;
        $this->engine = new Engine($power);
    }
}

$myCar = new Car( model: "Saturn VUE",  power: 160);
```

Плюсы

- Скрывает отношения использования объекта от глаз клиента.
- Делает API использования класса более простым.

Минусы

- Отношение жесткое (один объект должен знать конкретный тип и иметь доступ к функции создания другого объекта).

Агрегация

Объект А получает ссылку на объект В

```php
class Engine
{
    private $power;

    public function __construct($power)
    {
        $this->power = $power;
    }
}

class Car
{
    private $model;
    private $engine;

    public function __construct($model, $engine)
    {
        $this->model = $model;
        $this->engine = $engine;
    }
}

$myCar = new Car( model: "Saturn VUE", new Engine( power: 160));
```

Плюсы

- Более слабая связанность между объектом и его клиентом. Объекту не нужно знать, как именно создавать другой объект.
- Большая гибкость.

Минусы

- Выставление наружу деталей реализации, увеличение сложности в работе клиентов.
- «Целое» не может самостоятельно заменить «составную часть».

Различие между композицией и агрегацией

	Композиция	Агрегация
Жизненный цикл	Дочерний объект живет столько же времени, сколько и родительский	Объекты живут независимо друг от друга
Контроль над зависимостями	Владелец контролирует создание и удаление своих частей	Владелец просто использует другие объекты без контроля над их созданием и удалением
Зависимость	Объект жестко привязан к своему владельцу	Связь менее жесткая, так как объект может принадлежать нескольким владельцам одновременно
Уровень абстракции	Часто используется для реализации внутренней структуры объекта	Применяют для объединения нескольких независимых сущностей

Инверсия управления и внедрение зависимостей

Inversion of Control (IoC) — Инверсия управления

Если говорить максимально просто: **IoC — это принцип, при котором ваш код не управляет зависимостями сам, а получает их готовыми извне.**

Вместо того чтобы говорить: *"Мне нужен этот объект, я создам его сам"*, вы говорите: *"Мне нужен этот объект, пусть кто-нибудь другой мне его передаст"*.

1. Аналогия из жизни 🍕

Без IoC (Вы всё контролируете сами)

Вы хотите пиццу.

1. Вы покупаете тесто.
2. Вы покупаете сыр.
3. Вы сами печёте в своей духовке.
4. Вы сами доставляете её себе на стол.

Проблема: Если захотите суши вместо пиццы, придётся менять весь процесс (покупать рис, рыбу и т.д.). Вы сильно зависимы от конкретных ингредиентов.

С IoC (Вы заказываете в ресторане)

Вы хотите пиццу.

1. Вы приходите в ресторан (Фреймворк/Контейнер).
2. Говорите: *"Мне нужна еда"*.
3. Ресторан сам решает, где взять ингредиенты, кто их приготовит и как подаст.
4. Вам приносят готовое блюдо.

Преимущество: Вам не важно, как именно приготовили пиццу. Завтра ресторан может сменить поставщика сыра, а для вас ничего не изменится.

Инверсия управления и внедрение зависимостей

Основная цель инверсии управления (Inversion of Control, IoC) — отделить логику создания объектов и управления их жизненным циклом от бизнес-логики приложения.

Управление созданием зависимостей передается внешнему компоненту.

Типы IoC:

- **Внедрение зависимостей** (Dependency Injection, DI). Зависимости передаются объекту извне.
- **Service Locator**. Объект сам запрашивает зависимости через глобальный реестр.
- **Event-based**. Управление происходит через события.

Внедрение зависимостей

Dependency injection (DI) - передача зависимостей в класс снаружи. Это инструмент для создания чистого и гибкого кода, помогающий:

- Разделить ответственность между объектами.
- Сделать код более модульным и тестируемым.
- Упростить замену реализаций зависимостей.

Используется во многих современных PHP-фреймворках.

Внедрение зависимостей

- Хорошая функция получает нужные значения через аргументы, хороший класс **получает обязательные зависимости через конструктор**:
    - Нельзя забыть передать зависимость при создании объекта
    - По сигнатуре конструктора легко узнать, какие зависимости нужны
    - Можно создавать несколько объектов с разными настройками
    - В качестве зависимости можно передавать не только определенный класс, но и его потомков с измененным поведением
- Необязательные зависимости передаются через методы-сеттеры

Ручное внедрение зависимостей

```
class EmailSender {
    private $mailService;

    public function __construct(MailService $mailService) {
        $this->mailService = $mailService;
    }

    public function sendEmail($to, $subject, $body) {
        $this->mailService->send($to, $subject, $body);
    }
}
```

```
$mailService = new MailService();
$emailSender = new EmailSender($mailService);
$emailSender->sendEmail('user@example.com', 'Subject', 'Body');
```

Внедрение зависимостей через DI контейнер

```
use DI\ContainerBuilder;
                              //Контейнер PHP-DI
$containerBuilder = new ContainerBuilder();
$container = $containerBuilder->build();

// Регистрация классов
$container->set('MailService', function () {
    return new MailService();
});

$container->set('EmailSender', function ($container) {
    return new EmailSender($container->get('MailService'));
});
```

```
$emailSender = $container->get('EmailSender');
$emailSender->sendEmail('user@example.com', 'Subject', 'Body');
```

- **Управление зависимостями:** Контейнер автоматически разрешает все зависимости, даже если они вложены.
- **Модульность:** Упрощает разделение кода на независимые компоненты.
- **Тестирование:** Легко заменять реальные зависимости на мок-объекты для тестирования.
- **Гибкость:** Можно легко изменить реализацию зависимости без изменения основного кода.

Autowiring

Autowiring (автоматическое связывание) в DI контейнере — механизм, который автоматически создает объекты и их зависимости на основе типов параметров в конструкторах классов.

- **Автоматизация:** Не нужно явно регистрировать все зависимости вручную. PHP-DI автоматически создаст необходимые объекты.
- **Чистый код:** Код становится более чистым и понятным.
- **Масштабируемость:** При добавлении новых зависимостей PHP-DI автоматически их обрабатывает.

Autowiring с PHP-DI

2. Структура проекта

```
1  project/
2  ├── src/
3  │   ├── Database.php
4  │   ├── UserRepository.php
5  │   ├── UserService.php
6  │   └── UserController.php
7  ├── config.php
8  └── index.php
```

3. Классы с зависимостями

Обратите внимание: зависимости объявляются через типизацию в конструкторе. Это ключевое условие для работы autowiring.

`src/Database.php`

php

```php
<?php
namespace App;

class Database {
    public function connect(): string {
        return "Подключено к БД";
    }

    public function query(string $sql): array {
        return ["result" => $sql];
    }
}
```

Autowiring с PHP-DI

src/UserRepository.php

```php
<?php
namespace App;

class UserRepository {
    // PHP-DI автоматически создаст Database и передаст сюда
    public function __construct(
        private Database $db
    ) {}

    public function findById(int $id): array {
        $this->db->connect();
        return $this->db->query("SELECT * FROM users WHERE id = $id");
    }
}
```

Autowiring с PHP-DI

`src/UserService.php`

php

```php
<?php
namespace App;

class UserService {
    // PHP-DI автоматически создаст UserRepository
    public function __construct(
        private UserRepository $repo
    ) {}

    public function getUser(int $id): array {
        return $this->repo->findById($id);
    }
}
```

Autowiring с PHP-DI

src/UserController.php

php

```php
<?php
namespace App;

class UserController {
    // PHP-DI автоматически создаст UserService
    public function __construct(
        private UserService $service
    ) {}

    public function show(int $id): string {
        $user = $this->service->getUser($id);
        return "Пользователь найден: " . json_encode($user);
    }
}
```

Autowiring с PHP-DI

4. Настройка контейнера (`config.php`)

php

```php
<?php
use DI\ContainerBuilder;

$builder = new ContainerBuilder();

// 1. Включаем autowiring (по умолчанию true, но явно укажем)
$builder->useAutowiring(true);

// 2. Включаем аннотации/атрибуты (если будете использовать)
$builder->useAttributes(true);

// 3. Путь для кэша в production (опционально)
// $builder->enableCompilation(__DIR__ . '/cache');

// 4. Добавляем определения для интерфейсов (если есть)
$builder->addDefinitions([
    // Пример: когда просят интерфейс, даём конкретный класс
    // App\CacheInterface::class => \DI\create(App\RedisCache::class),
]);

return $builder->build();
```

Autowiring с PHP-DI

5. Использование (`index.php`)

php

```php
<?php
require_once 'vendor/autoload.php';

// Создаём контейнер с конфигурацией
$container = require 'config.php';

// ЗАПРОС ГЛАВНОГО КЛАССА
// PHP-DI сам создаст всю цепочку:
// UserController → UserService → UserRepository → Database
$controller = $container->get(App\UserController::class);

// Работаем с объектом
echo $controller->show(123);
// Вывод: Пользователь найден: {"result":"SELECT * FROM users WHERE id = 123"}
```

6. Что происходит внутри? (Схема)

```
1  Вы запрашиваете: $container->get(UserController::class)
2                          ↓
3  PHP-DI смотрит конструктор UserController
4                          ↓
5  Видит зависимость: UserService
6                          ↓
7  PHP-DI создаёт UserService (рекурсия)
8                          ↓
9  Видит зависимость: UserRepository
10                         ↓
11 PHP-DI создаёт UserRepository (рекурсия)
12                         ↓
13 Видит зависимость: Database
14                         ↓
15 PHP-DI создаёт Database (нет зависимостей ✅)
16                         ↓
17 Собирает всё обратно вверх по цепочке
18                         ↓
19 Вы получаете готовый UserController
```

Autowiring с PHP-DI

Компонент	Код
Включение autowiring	`$builder->useAutowiring(true)`
Получение объекта	`$container->get(Class::class)`
Интерфейсы	Требуют явной настройки в `addDefinitions()`
Параметры	Требуют `constructorParameter()`
Production	Включать `enableCompilation()`

Autowiring с PHP-DI

Autowiring в PHP-DI позволяет вам:

1. Не писать конфигурацию для каждого класса.
2. Менять зависимости без изменения кода.
3. Фокусироваться на бизнес-логике, а не на сборке объектов.

Просто типизируйте аргументы конструктора, и PHP-DI сделает всё остальное автоматически! 🚀