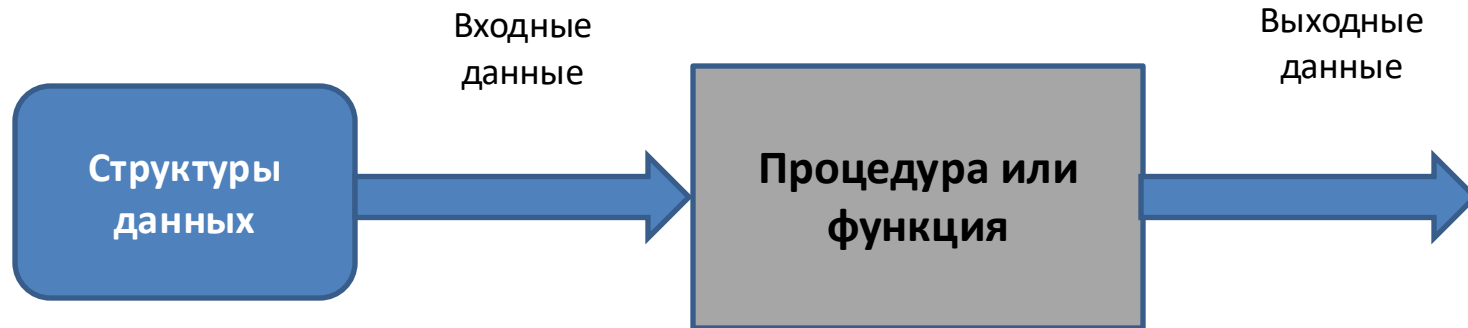


2. Основные понятия и механизмы ООП

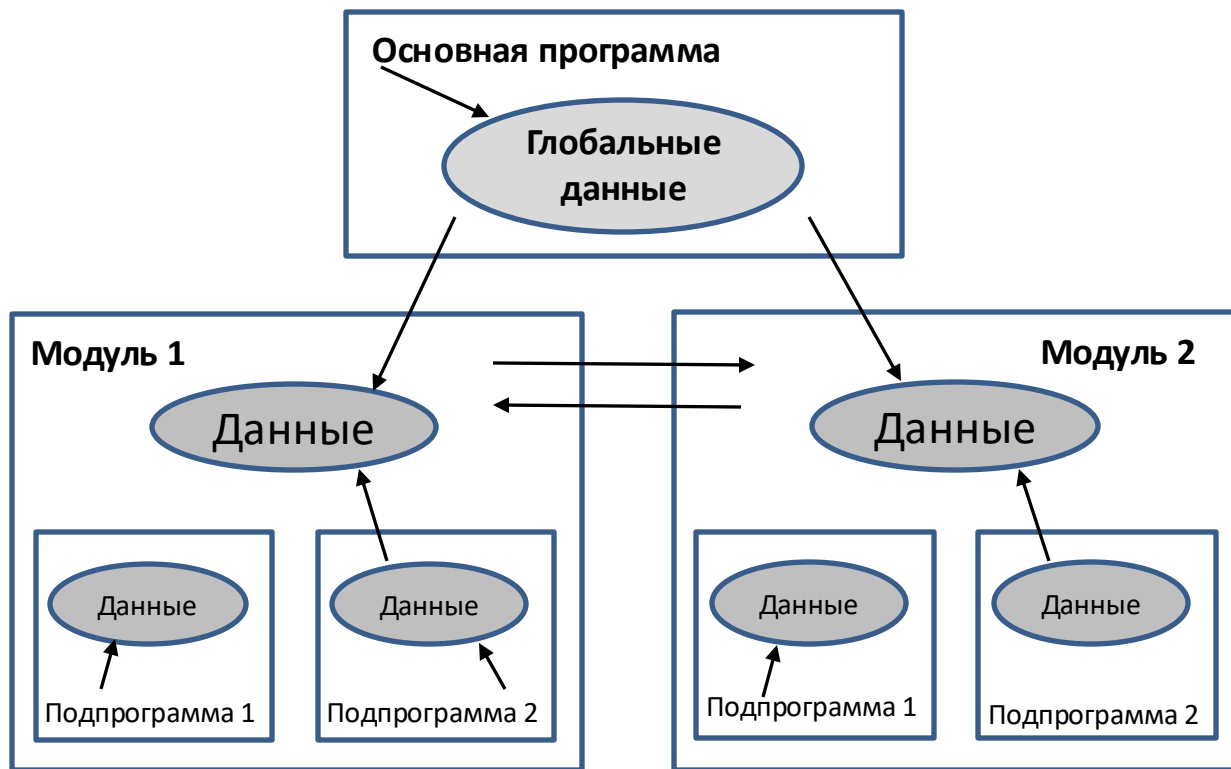
Процедурный подход

- Сложные программы делятся на простые фрагменты с помощью подпрограмм.
- Данные и операции – логически разные сущности.
 - Данные составляют состояние системы
 - Алгоритмы определяют её поведение.



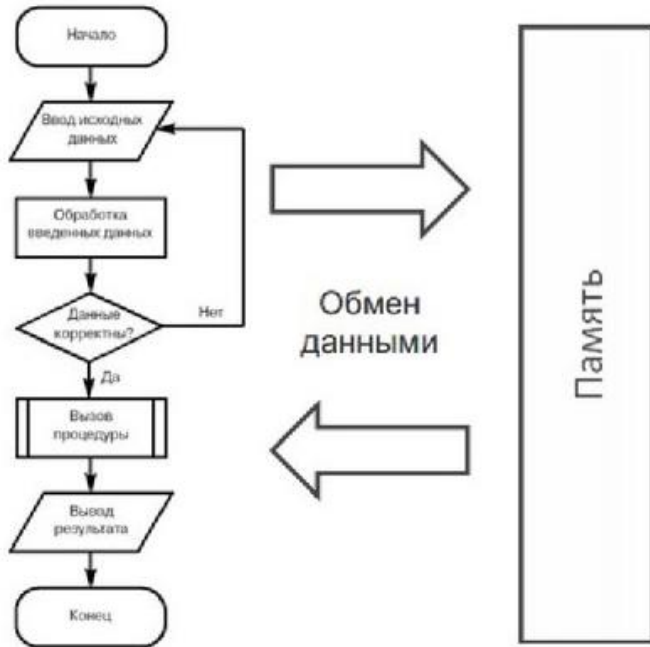
Модульный подход: инкапсуляция и сокрытие данных

Данные и обрабатывающие их функции локализуются (инкапсулируются) в модулях и защищаются от внешних воздействий с помощью интерфейсов.



Две парадигмы программирования

Процедурный подход

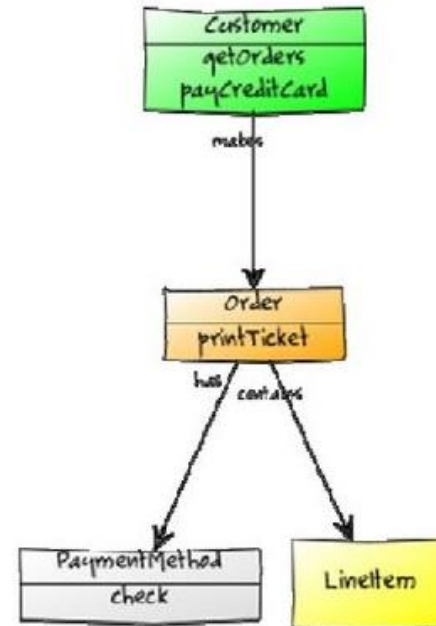


Программа – алгоритм последовательного вызова процедур изменения данных в памяти.

Программа = Алгоритмы + Структуры данных

(Н. Вирт)

Объектно-ориентированный

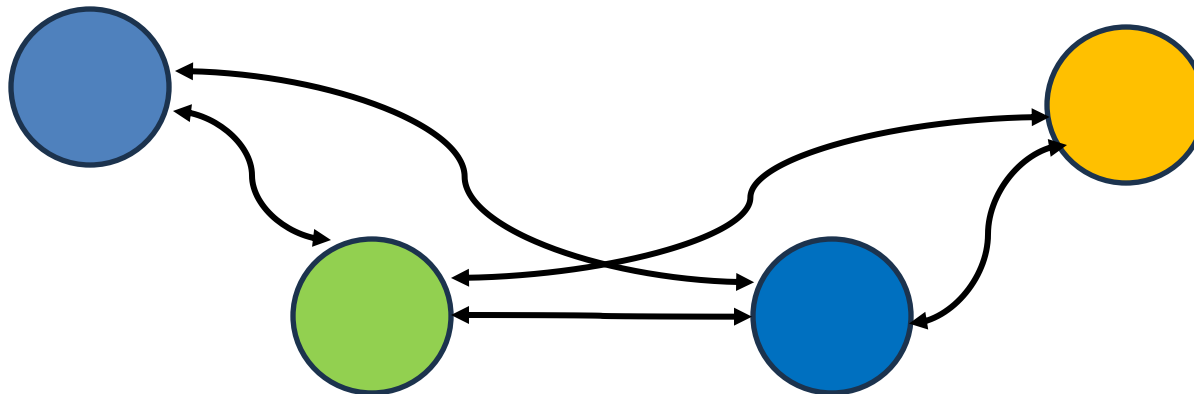


Программа – взаимодействие объектов, компонентов, отсылка и обработка событий.

Приложение строится из «кирпичей» - объектов

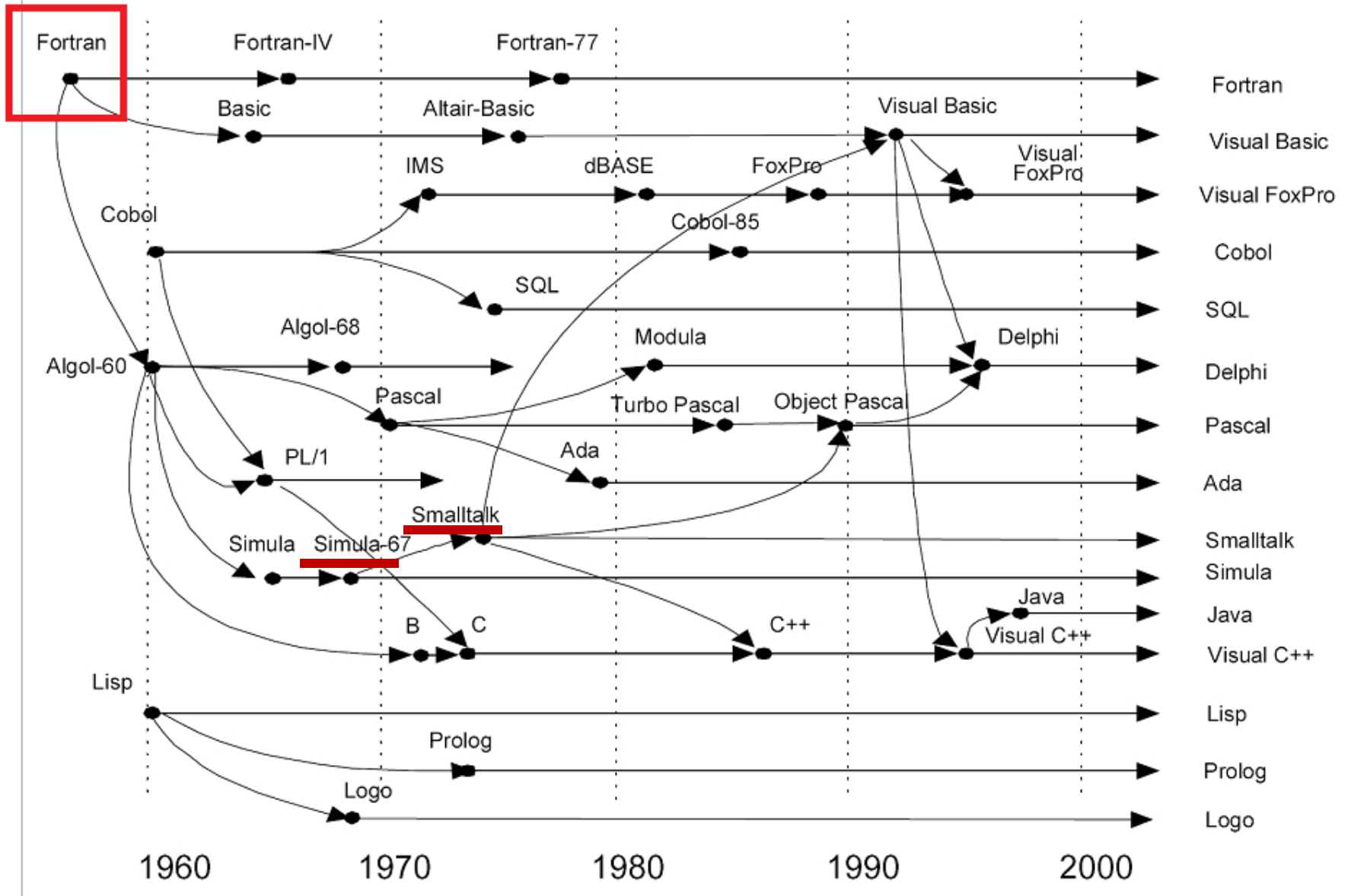
Суть ООП

- 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.



Первые реализации ООП

ГЕНЕАЛОГИЧЕСКОЕ ДЕРЕВО ЯЗЫКОВ ПРОГРАММИРОВАНИЯ



Simula 67 (SIMUlation LAnguage)

Разработан в конце 1960-х в Норвежском вычислительном центре (Осло)

- Кристен Нюгард — компьютерный ученый, специалист по моделированию и системному анализу
- Оле-Джон Дейч — программист и математик

Язык Simula оказал влияние на Smalltalk, C++, Java, Python, Ruby

Simula 67

Основные цели

- Моделирование реальных процессов и систем. Описание систем через события, процессы и их взаимодействие.
- Инкапсуляция и повторное использование кода.
- Создание новой парадигмы программирования, позволяющей мыслить в терминах объектов и их поведения.

Simula 67

Концепции и понятия

- **Классы и объекты.** Объекты имеют свои данные и методы. Классы - шаблоны для создания объектов
- **Инкапсуляция.** Скрытие внутренней реализации объекта от внешнего мира.
- **Наследование.** Создание новых классов на основе существующих, с добавлением или изменением функциональности.
- **Сообщения.** Объекты взаимодействуют друг с другом через отправку сообщений

```
CLASS Point(xCoord, yCoord);
  REAL xCoord, yCoord;

BEGIN
  TEXTUAL OUT("Creating a point with coordinates (", xCoord, ", ", yCoord, ")");

  PROCEDURE Move(newX, newY);
    REAL newX, newY;
  BEGIN
    xCoord := newX;
    yCoord := newY;
    OUT("Point moved to (", xCoord, ", ", yCoord, ")");
  END;

  REAL PROCEDURE Distance(other);
    REF(Point) other;
  BEGIN
    REAL dx, dy;
    dx := xCoord - other.xCoord;
    dy := yCoord - other.yCoord;
    RETURN SQRT(dx * dx + dy * dy);
  END;
END;

BEGIN
  REF(Point) p1, p2;
  p1 := NEW Point(0, 0); ! Создание первой точки с координатами (0, 0)
  p2 := NEW Point(3, 4); ! Создание второй точки с координатами (3, 4)

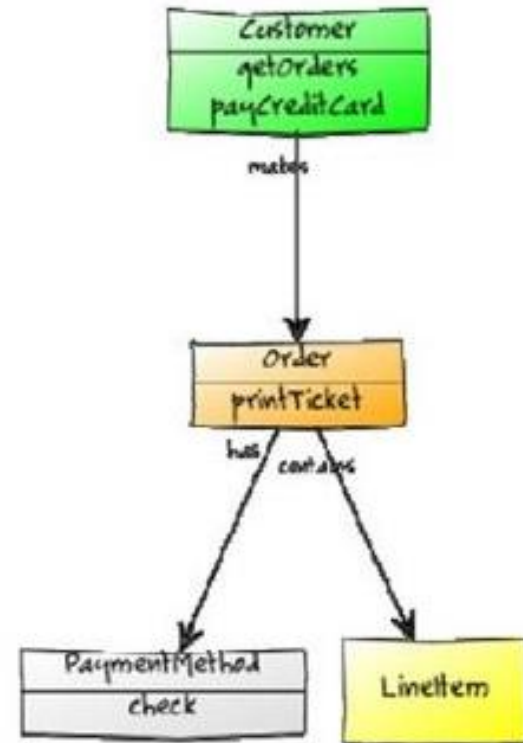
  OUT("Distance between p1 and p2 is ", p1.Distance(p2)); ! Вычисление расстояния

  p1.Move(1, 1); ! Перемещение первой точки
  OUT("New distance between p1 and p2 ↓", p1.Distance(p2)); ! Вычисление нового
END.
```

SmallTalk

Первый полностью объектно-ориентированный язык

- Всё в языке является объектом
- Взаимодействие между объектами только через сообщения



Создатель SmallTalk



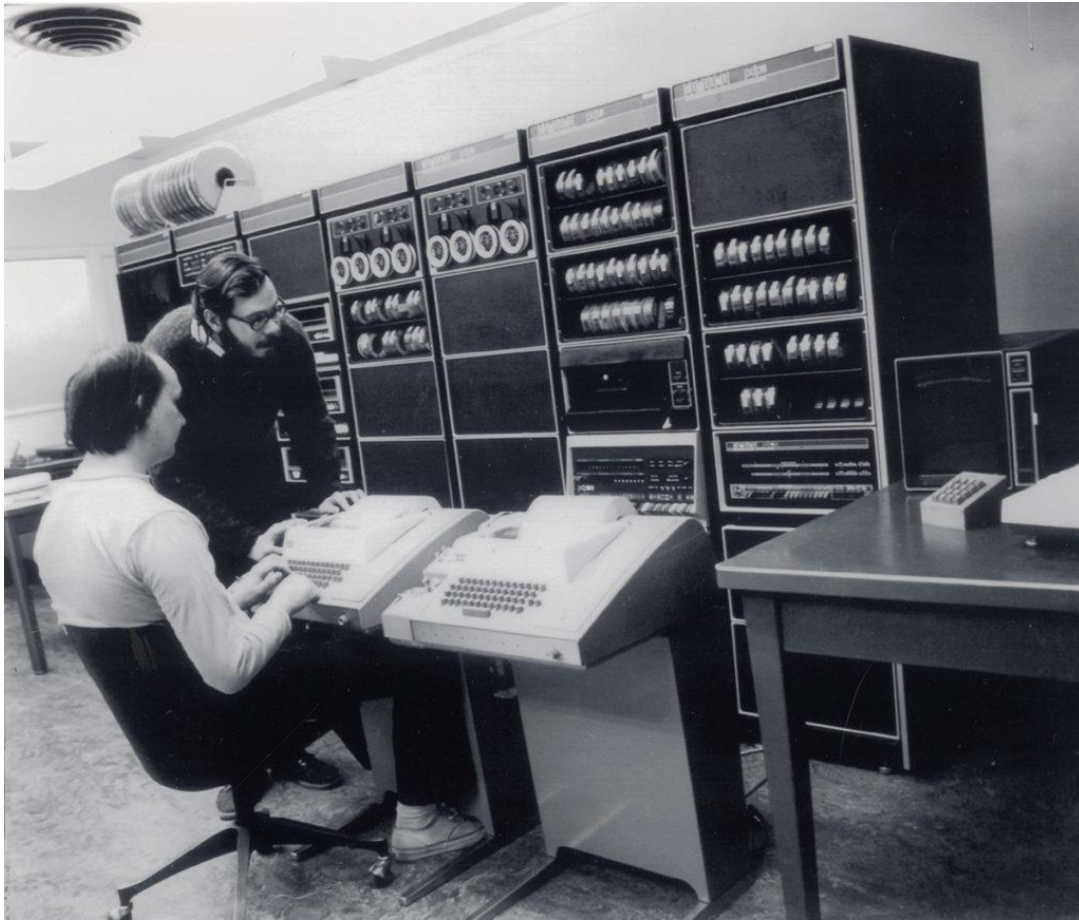
Алан Кёртис Кей (1940 г.р.)

- Джазовый музыкант
- Математическое и биологическое образование

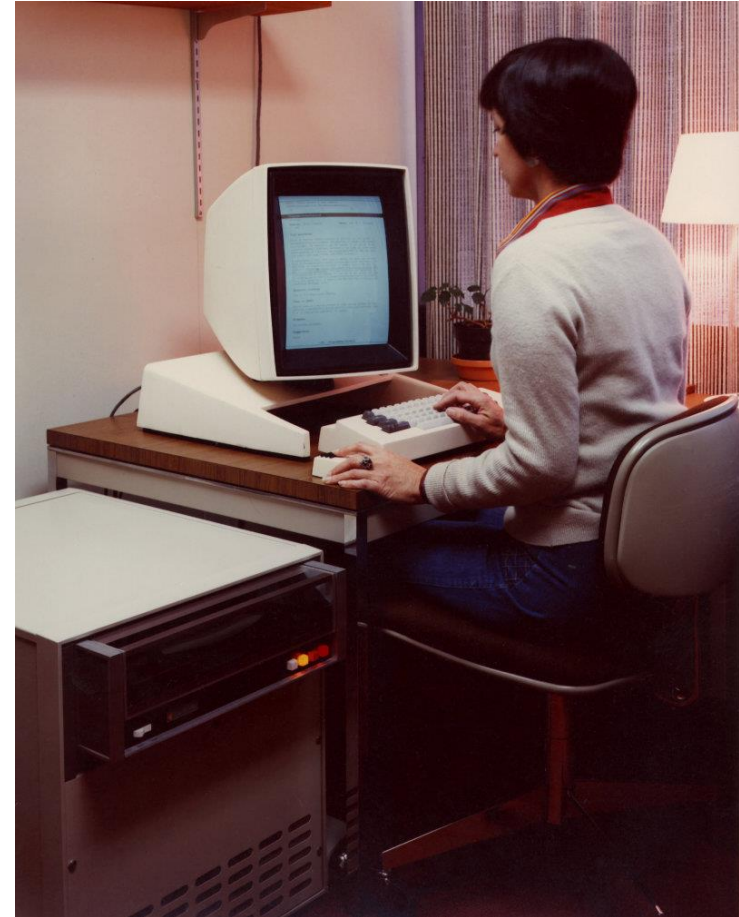
В 1970-е работал в Xerox PARC (Palo Alto Research Center) над персональным компьютером будущего Dynabook - прототипом ноутбука и планшета.



Компьютер из будущего в 1970-х

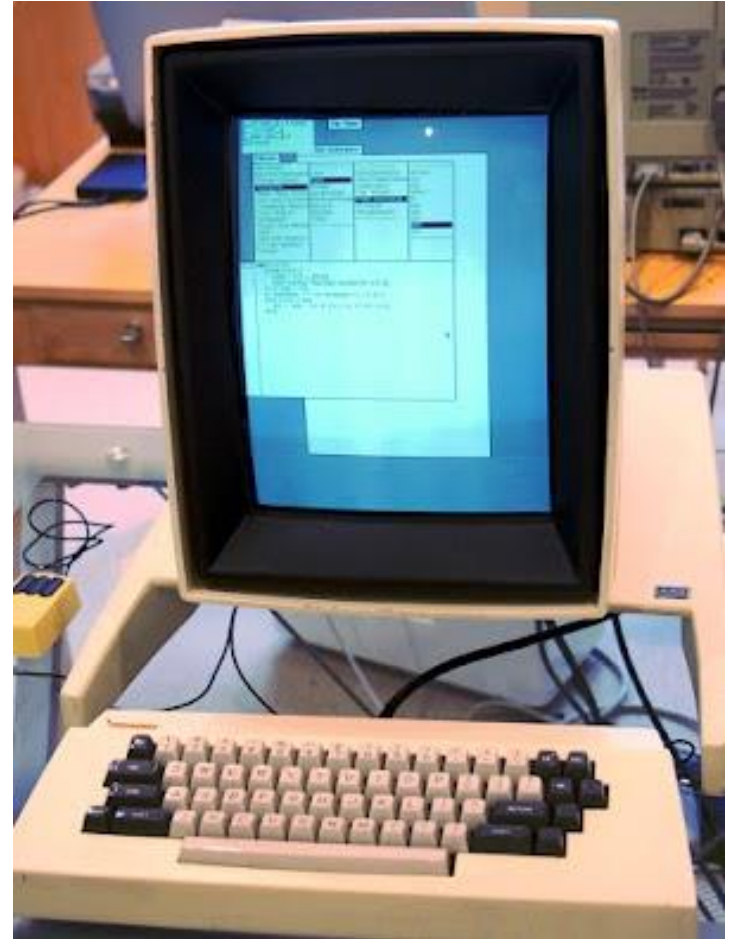
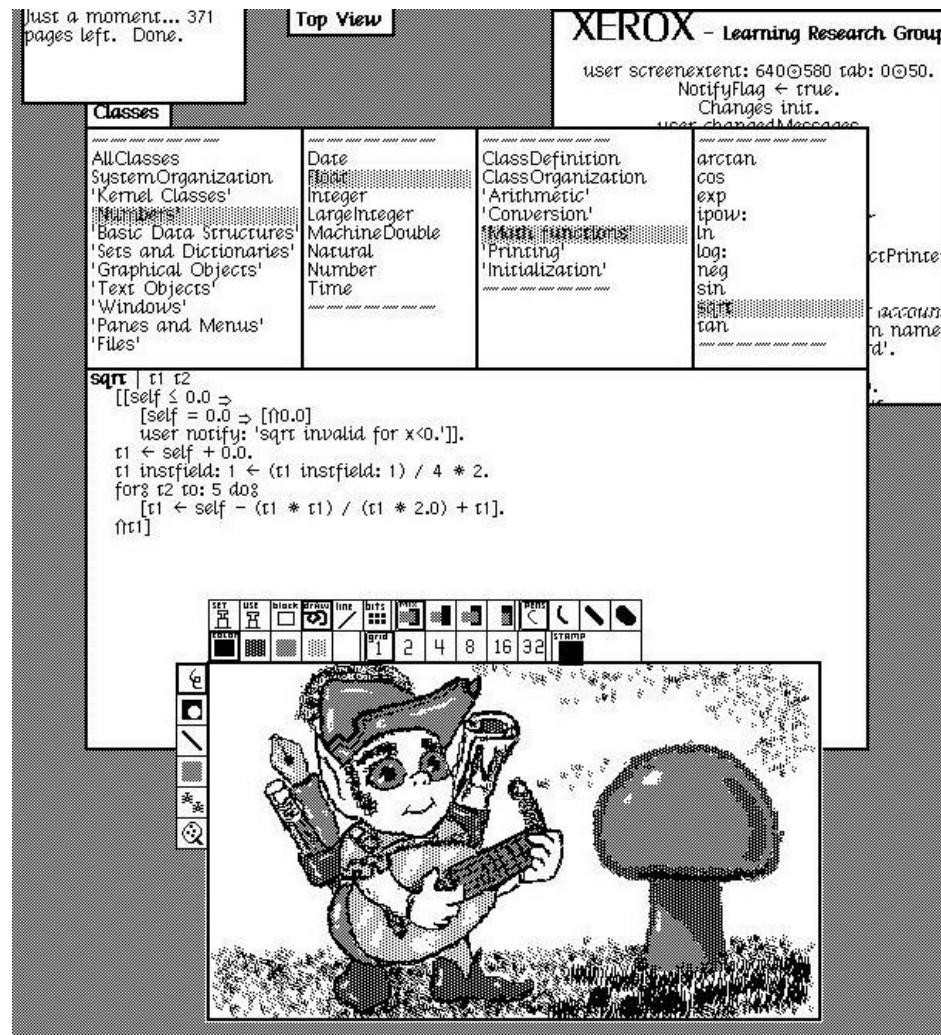


За терминалом создатели UNIX:
Кен Томпсон и Деннис Ритчи



Xerox Alto (1973 год)

Первая интерактивная графическая среда разработки И ВЫПОЛНЕНИЯ



Xerox Alto (1973 год)

SmallTalk-76

Особенности SmallTalk

- Всё является объектом, даже базовые типы данных
- Сообщения вместо вызовов функций
- Динамическая типизация
- Блоки кода, которые можно передавать как параметры.
- Позднее связывание и динамическая привязка (возможность программ развиваться и адаптироваться к изменениям во время их выполнения)
- Интерактивность и рефлексия (в среде разработки можно изменять код и сразу видеть результат)
- Выполнение через байт-код и виртуальную машину

"Определение класса Point"

```
Object subclass: #Point
  instanceVariableNames: 'x y'
  classVariableNames: ''
  package: 'Example'
```

"Инициализация точки"

```
Point>>initializeX: xValue Y: yValue
  x := xValue.
  y := yValue.
```

"Метод для перемещения точки"

```
Point>>moveToX: newX Y: newY
  x := newX.
  y := newY.
```

"Метод для вычисления расстояния до другой точки"

```
Point>>distanceTo: anotherPoint
  | dx dy |
  dx := x - anotherPoint x.
  dy := y - anotherPoint y.
  ^ (dx squared + dy squared) sqrt
```

"Создание объектов и их использование"

```
| p1 p2 distance |
```


```
p1 := Point new initializeX: 0 Y: 0. "Создание первой точки с координатами (0, 0)"
```

```
p2 := Point new initializeX: 3 Y: 4. "Создание второй точки с координатами (3, 4)"
```

```
distance := p1 distanceTo: p2. "Вычисление расстояния между точками"
```

```
Transcript show: 'Distance between p1 and p2 is '; show: distance; cr.
```

```
p1 moveToX: 1 Y: 1. "Перемещение первой точки"
```

```
distance := p1 distanceTo: p2. "Вычислен  нового расстояния"
```

```
Transcript show: 'New distance between p1 and p2 is '; show: distance; cr.
```

Современный подход к ООП

Синтез идей и подходов из разных языков

- Simula - основы классов и наследования
- Smalltalk - чистый объектный подход и сообщения
- C++, Java, Python и др. - статическая типизация, автоматическое управление памятью, метапрограммирование, функциональное программирование и асинхронность.

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

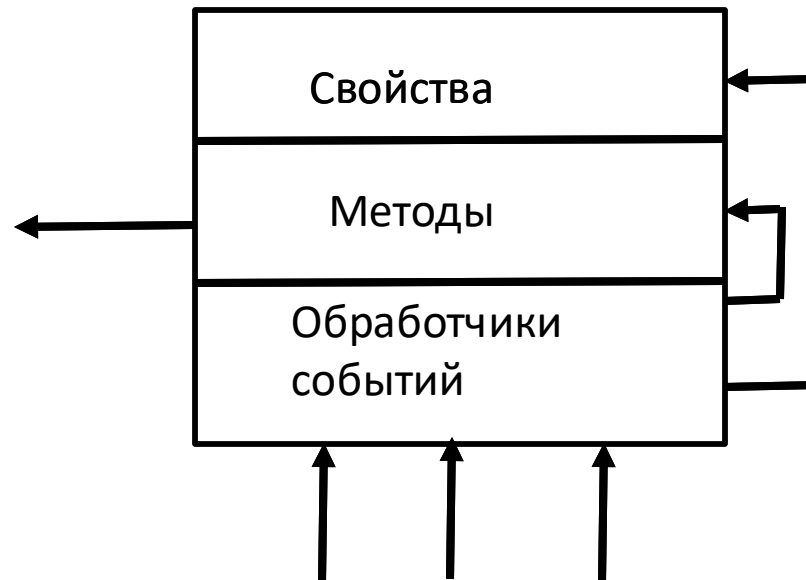
Объекты и события

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



Объектно-ориентированный API

```
pwsh
Файл Правка Вид Поиск Терминал Справка
→ ~ pwsh -- INSERT --
PowerShell 7.4.6
PS /home/andrey> "Строка из четырех слов".ToUpper()
СТРОКА ИЗ ЧЕТЫРЕХ СЛОВ
PS /home/andrey> "Два слова".
Length                LastIndexOfAny        ToInt32
Clone                 Normalize              ToInt64
CompareTo             PadLeft                ToLower
Contains              PadRight              ToLowerInvariant
CopyTo                Remove                 ToSByte
EndsWith              Replace                ToSingle
EnumerateRunes        ReplaceLineEndings    ToString
Equals                Split                  ToType
GetEnumerator          StartsWith             ToUInt16
GetHashCode           Substring              ToUInt32
GetPinnableReference ToBoolean              ToUInt64
GetType               ToByte                 ToUpper
GetTypeCode           ToChar                 ToUpperInvariant
IndexOf               ToCharArray            Trim
IndexOfAny            ToDateTime             TrimEnd
Insert                ToDecimal              TrimStart
IsNormalized          ToDouble               TryCopyTo
LastIndexOf           ToInt16                Chars
PS /home/andrey>
```

Создание и использование объектов

```
vifm
Файл  Правка  Вид  Поиск  Терминал  Справка
1  let cat = {
2      name: "Мурзик",
3      say: function () {
4          | console.log(`Мяу! Меня зовут ${this.name}!`);
5      }
6  }
7  cat.say();
~
~
~
~/projects/mrsu-examples/PHP/05_PHP_OOP/00_object.js  jav...  12%  ≡ 1/8  ln :1
→ 05_PHP_OOP git:(main) node 00_object.js
Мяу! Меня зовут Мурзик!
→ 05_PHP_OOP git:(main) █                                -- INSERT --

TERMINAL /bin/zsh  ≡ 1 /1 ln
:term
```

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 АДД, классы для работы с файловой системой, ...)

Описание класса и создание объекта

```
<?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();
```

```
class Student:
    def get_full_name(self):
        | print("Полное имя: " + self.name + ' ' + self.last_name)

# ===== Create objects =====

student1 = Student()

student1.name = 'Сергей'
student1.last_name = 'Иванов'
student1.get_full_name()
```

Конструктор

Основное назначение – инициализация атрибутов объекта.

- **C++**, **Java**, **C#** – метод, совпадающий с именем класса
- **Visual Basic** – метод `New`
- **Object Pascal** – метод `Create`
- **PHP** – метод `__construct`
- **Python** – метод `__init__`

Конструктор в Python

Метод `__init__()`

Основное назначение – инициализация атрибутов объекта.

```
class Student:
    def __init__(self, name, last_name):
        self.name = name
        self.last_name = last_name

    def get_full_name(self):
        print("Полное имя: " + self.name + ' ' + self.last_name)

student1 = Student('Сергей', 'Иванов')
student1.get_full_name()
```

Конструктор в PHP

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 в PHP

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 в PHP

Начиная с PHP 8, свойства можно объявлять прямо в конструкторе.

```
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 в PHP

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

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

Магические методы в 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, Ива
```

Именованние 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 в PHP

- 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
```

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 в PHP

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

Статические свойства и методы

```
<?php
1 class Counter {
2     public static $count = 0; // Статическое свойство
3
4     public function increment() {
5         |     self::$count++; // Увеличиваем счётчик
6     }
7
8     public static function getCount() {
9         |     return self::$count; // Возвращаем текущее значение счётчика
10    }
11 }
12
13 Counter::$count = 100; // Устанавливаем начальное значение счётчика
14
15 $counter1 = new Counter();
16 $counter1->increment(); // Увеличили счётчик до 101
17
18 $counter2 = new Counter();
19 $counter2->increment(); // Увеличили счётчик до 102
20
21 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.

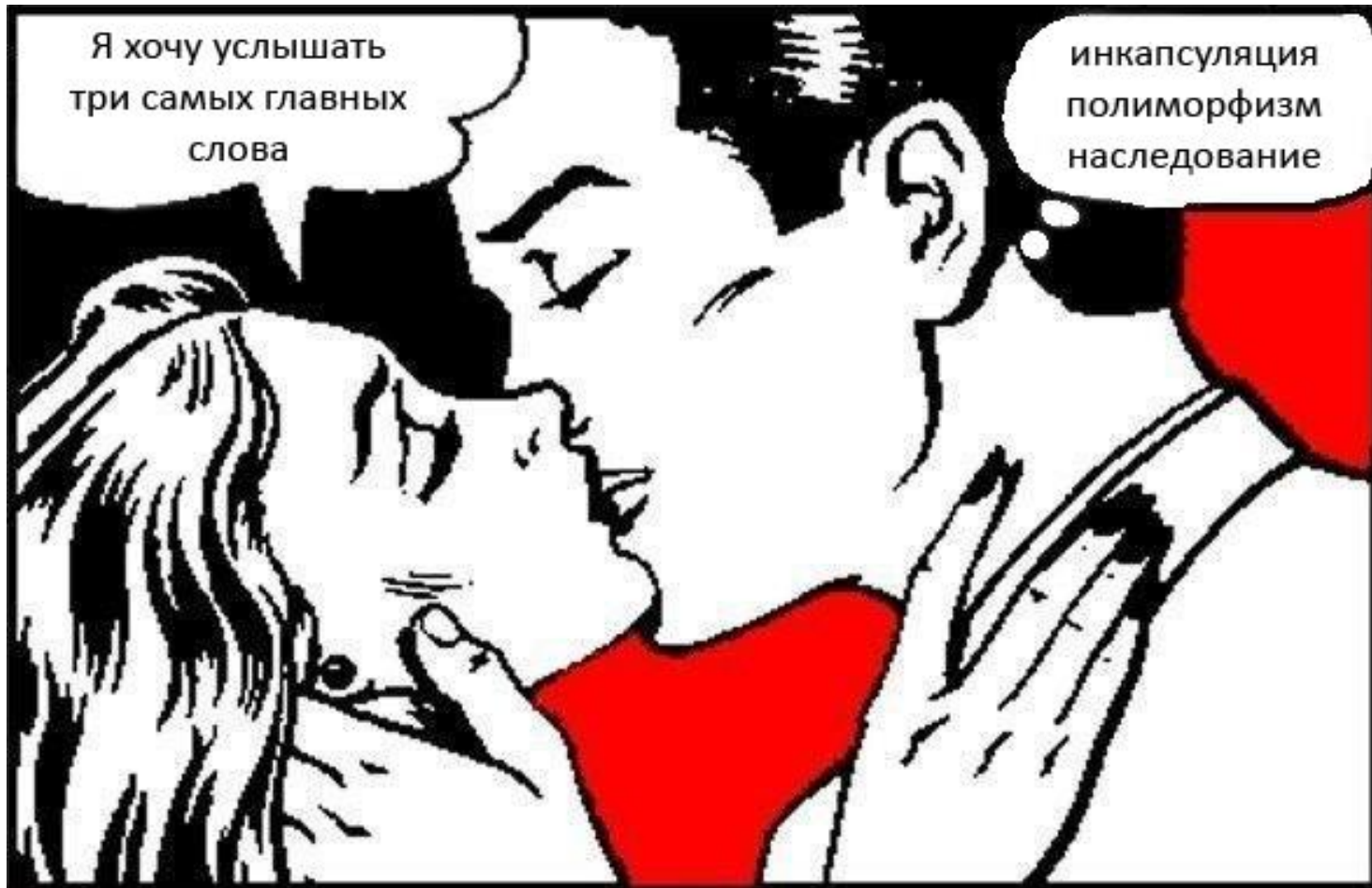
Причины создания классов

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

Виды объектов:

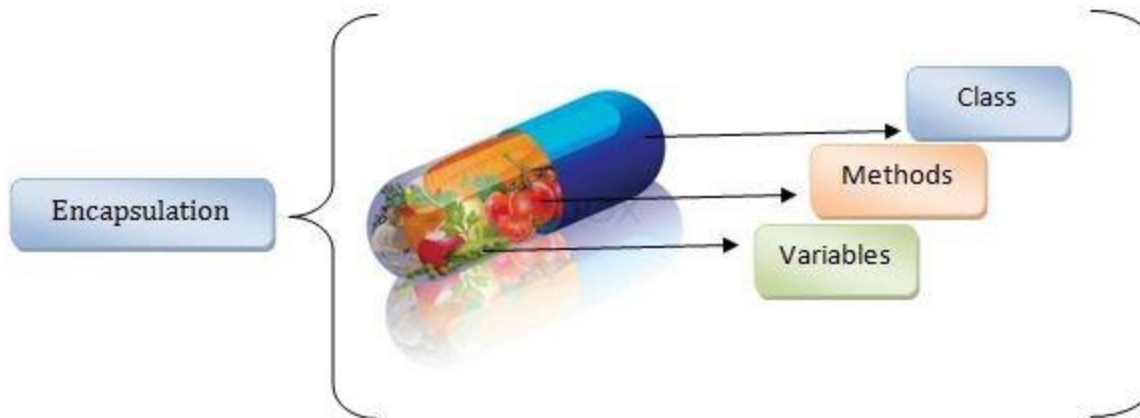
- Объекты-сущности
- Объекты-значения

Основные принципы ООП



Инкапсуляция и сокрытие данных

Инкапсуляция - упаковка данных и функций в один компонент.



Соккрытие 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**.

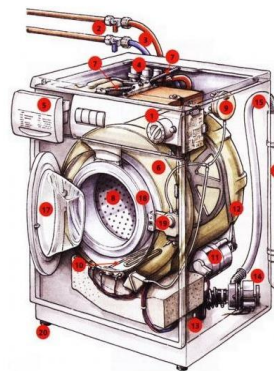
У этого термина существуют и другие значения, см. [Инкапсуляция](#).

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

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



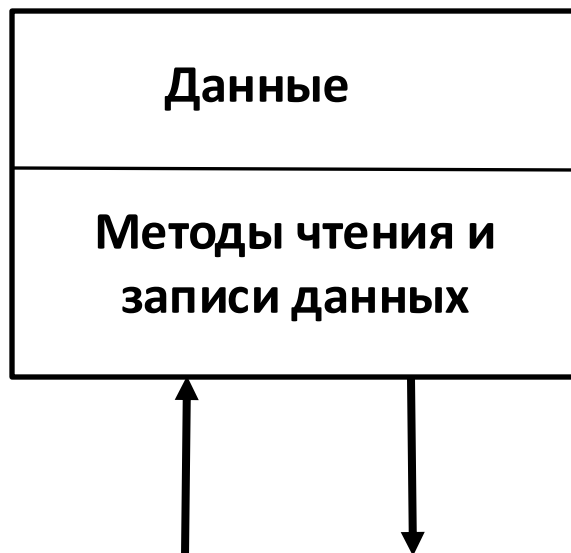
Интерфейс

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 прочесть названия публичных методов.
- Отдельные части программы (объекты) можно разрабатывать и тестировать независимо друг от друга.

Соккрытие данных – реализация в ЯП

Object Pascal

```
Type
  TTimePeriod = class
    function GetProperty : integer;
    procedure SetProperty(NewValue : integer);
    property Hours : integer read GetProperty write SetProperty;
  End;

var
  TimePeriod : TTimePeriod;
  h: integer;

  TimePeriod := TTimePeriod.Create;
  TimePeriod.Hours := 24;
  h := TimePeriod.Hours;
```

Соккрытие данных – реализация в ЯП

C#

```
class TimePeriod
{
    private double seconds;
    public double Hours
    {
        get { return seconds / 3600; }
        set { seconds = value * 3600; }
    }
}

class Program
{
    static void Main()
    {
        TimePeriod t = new TimePeriod();
        t.Hours = 24;
        System.Console.WriteLine("Time in hours: " + t.Hours);
    }
}
```

Механизм сокрытия данных в PHP

Области видимости свойств и методов класса в PHP:

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

```
class Person {
    private $name;

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

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

// Использование класса
$person = new Person();
$person->setName("Александр");
echo $person->getName(); // Выведет: Александр
```

Механизм сокрытия данных в Python

- По умолчанию все атрибуты классов и объектов публичные.
- Если название начинается с "_", то считается, что его можно использовать только внутри класса (это просто соглашение)
- Если название начинается с "__", то применяется искажение имени (name mangling)

```
class MyClass:
    __private_attr = "Я приватный атрибут"

    def __init__(self, value):
        self.__private_instance_attr = value

# Прямой доступ вызовет ошибку
obj = MyClass(30)
# print(obj.__private_instance_attr) # AttributeError

# Но можно обратиться через name mangling
print(obj._MyClass__private_instance_attr) # Вывод: 30
```

Методы-аксессоры в PHP

- `__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

Объект с динамической структурой данных

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

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

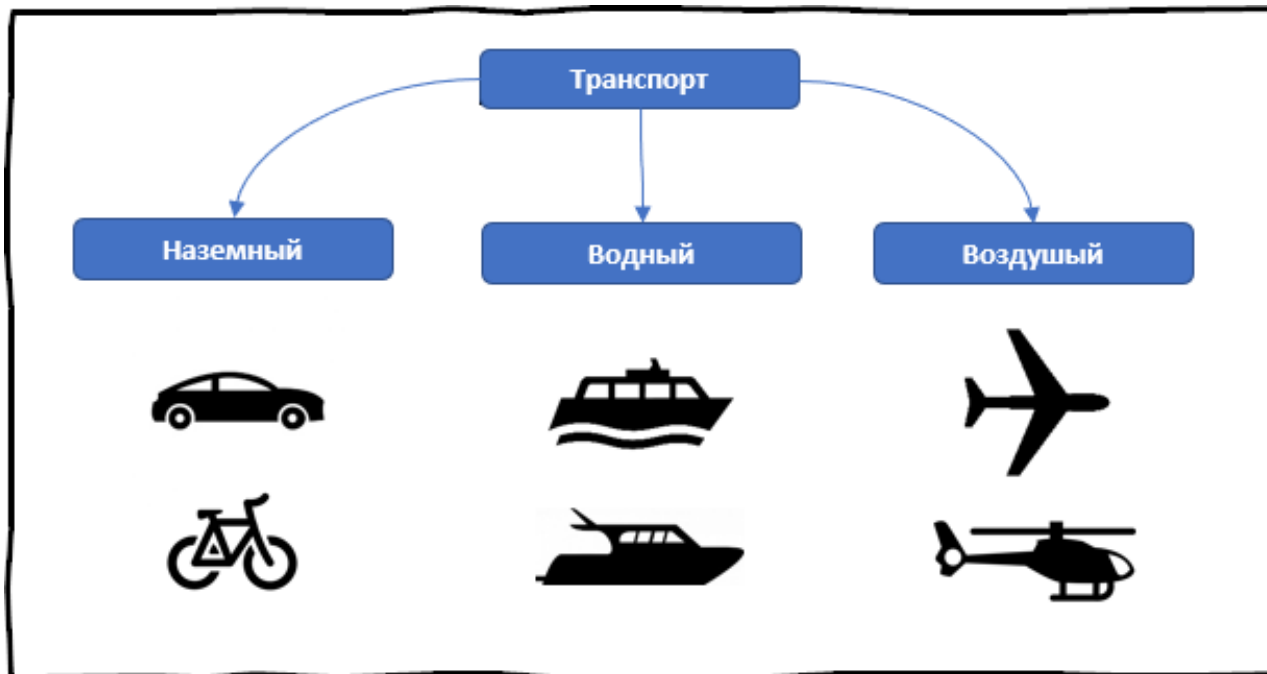
    public function __set($name, $value) {
        $this->data[$name] = $value;
    }
}

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

Наследование

Наследование

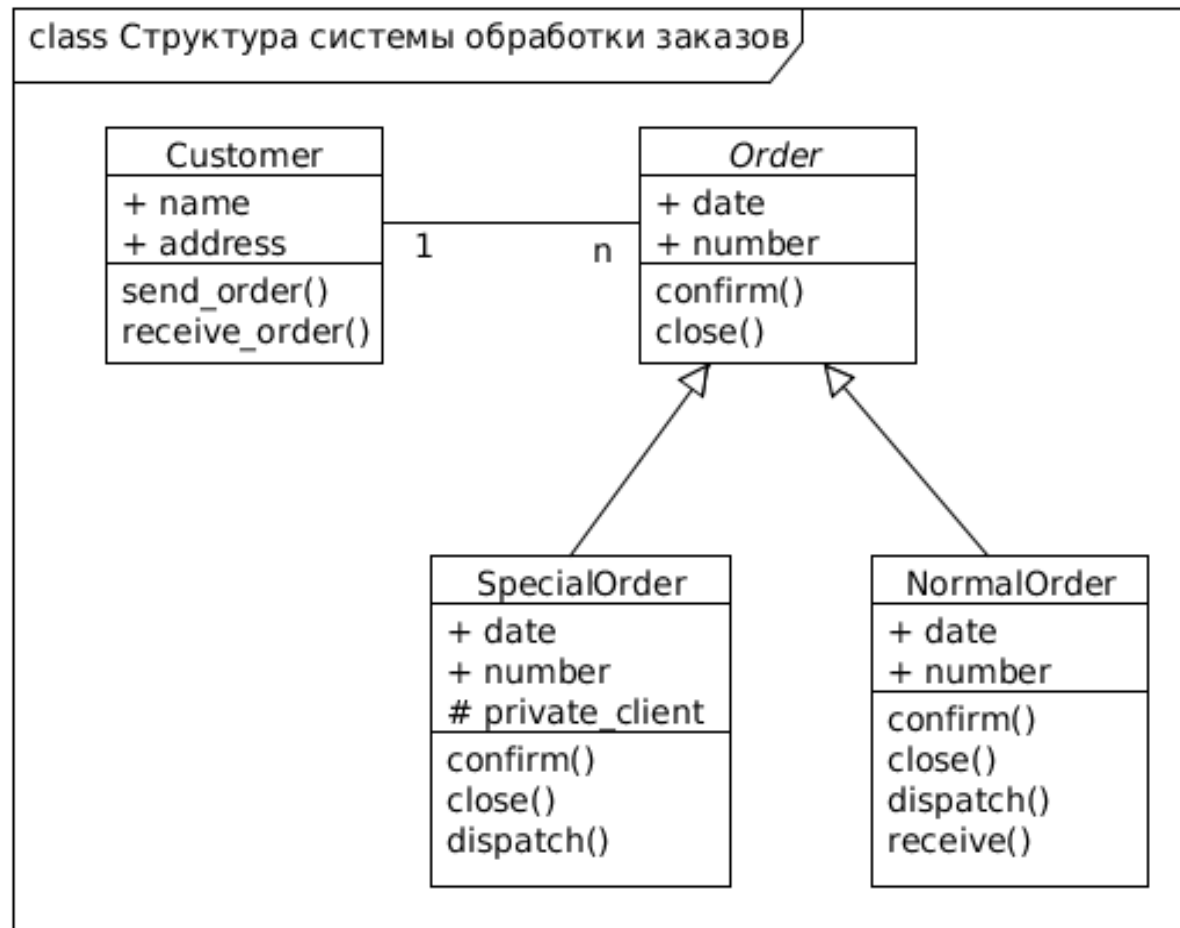
Наследование (иерархия) - возможность порождать один класс от другого с сохранением по наследству атрибутов и поведений от родительских классов, добавляя, при необходимости, новые свойства и методы.



Отношение “является”

Принцип наследования/расширения

Если вас *почти* устраивает какой-то класс, вы можете создать потомка и переопределить или добавить какую-то часть его функциональности.



Принцип наследования/расширения в PHP

- 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:

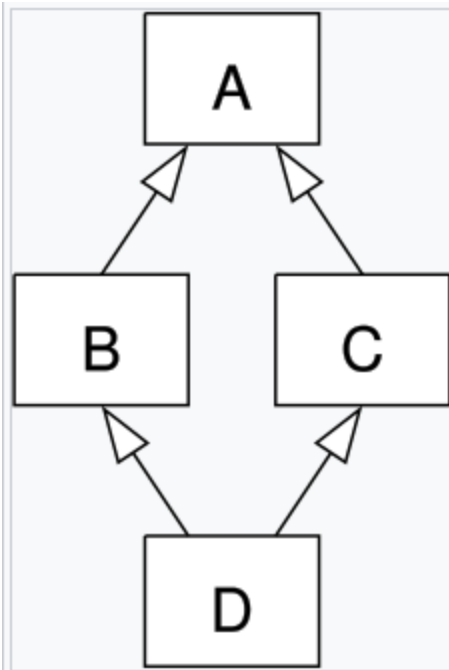
- Перекрыть метод родителя.
- Вызвать одноименный метод родителя через `parent::method()`

Множественное наследование

В C++, Python, Perl поддерживается **множественное наследование**, когда дочерний класс может иметь несколько родительских.

Языки Java, C#, Object Pascal, PHP этого не поддерживают.

Проблема ромбовидного наследования



```
class A:
    def greet(self):
        print("Привет из A")

class B(A):
    def greet(self):
        print("Привет из B")

class C(A):
    def greet(self):
        print("Привет из C")

class D(B, C):
    pass

d = D()
d.greet() # Какой метод будет вызван?
```

Абстрактные классы и методы

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

- Абстрактный класс может содержать как абстрактные методы (без реализации), так и обычные методы с реализацией.
- Все абстрактные методы должны быть реализованы в дочерних классах.
- Дочерние классы наследуют все свойства и методы абстрактного класса, включая реализованные методы.

Абстрактные классы и методы

```
abstract class Animal {
    protected $name;

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

    // Общий метод
    public function getName() {
        return $this->name;
    }

    // Абстрактный метод, который должны реализовать дочерние классы
    abstract public function makeSound();
}

// Дочерний класс
class Dog extends Animal {
    public function makeSound() {
        return "Woof";
    }
}

class Cat extends Animal {
    public function makeSound() {
        echo "Meow!\n";
    }
}
```

Абстрактные классы

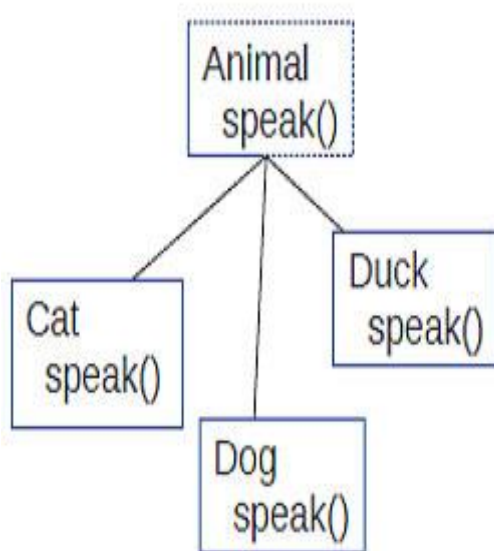
Абстрактные классы полезны в следующих случаях:

- Когда нужно определить общий интерфейс для нескольких классов.
- Когда нужно убедиться, что подклассы реализуют определенные методы.
- Когда нужно предоставить общую реализацию для некоторых методов, оставляя другие методы для реализации в подклассах.

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 - это один интерфейс для множества реализаций

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.

Позволяет писать обобщенные алгоритмы, которые могут работать с различными типами данных, не заботясь о деталях их реализации.

Ценность полиморфизма

- Позволяет писать обобщенные алгоритмы, которые могут работать с различными типами данных, не заботясь о деталях их реализации.
- Упрощает тестирование и отладку. Позволяет легко заменять объекты на мок-объекты (mock objects) или заглушки (stubs) во время тестирования.
- Позволяет работать с разными структурами данных через общий интерфейс. Например, функция может принимать любой итерируемый объект.

Интерфейсы

Интерфейсы

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



Что здесь?



Что здесь?





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

```
// Объявление интерфейса для системы логирования
interface Logger {
    public function log(string $message): void;
}

// Реализация интерфейса для записи логов в файл
class FileLogger implements Logger {
    public function log(string $message): void {
        file_put_contents('log.txt', $message . PHP_EOL, FILE_APPEND);
        echo "Logged to file: $message\n";
    }
}

// Реализация интерфейса для вывода логов в консоль
class ConsoleLogger implements Logger {
    public function log(string $message): void {
        echo "Logged to console: $message\n";
    }
}
```

Интерфейсы

```
// Сервис, который использует логгер
class UserService {
    private Logger $logger;

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

    public function createUser(string $username): void {
        $this->logger->log("Creating user: $username");
        // Логика создания пользователя
        echo "User $username created.\n";
    }
}

// Использование
$fileLogger = new FileLogger();
$consoleLogger = new ConsoleLogger();

$userServiceWithFileLogger = new UserService($fileLogger);
$userServiceWithFileLogger->createUser("john_doe");

$userServiceWithConsoleLogger = new UserService($consoleLogger);
$userServiceWithConsoleLogger->createUser("jane_doe");
```

Интерфейсы

Интерфейсы позволяют легко заменять реализации (например, переключаться между различными способами логирования), сохраняя при этом единый интерфейс взаимодействия.

Это делает код более гибким и тестируемым.

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

```
// Определяем трейт
trait SayHello {
    public function greet() {
        echo "Hello, ";
    }
}

trait SayWorld {
    public function farewell() {
        echo "World!";
    }
}

// Используем трейты в классе
class Greeter {
    use SayHello, SayWorld;

    public function welcome() {
        $this->greet();
        $this->farewell();
    }
}

// Создаем объект и используем методы из трейтов
$greeter = new Greeter();
$greeter->welcome(); // Выведет: Hello, World!
```

Примеси - зачем еще одно понятие?

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

В отличие от наследования, трейты не фиксируют структуру классов.

Примеси - зачем еще одно понятие?

Трейты особенно полезны, когда:

- Нужно повторно использовать код между классами, которые находятся в разных иерархиях наследования.
- Необходимо добавить определенную функциональность без создания лишних зависимостей.
- Хочется избежать множественного наследования, которое может привести к проблемам с управлением кодом.

Примеси

```
// Трейт для логирования
trait LoggerTrait {
    public function log($message) {
        echo "LOG: " . $message . PHP_EOL;
    }
}

// Класс пользователя
class User {
    use LoggerTrait;

    public function login() {
        $this->log("User logged in");
    }
}

// Класс платежной системы
class PaymentSystem {
    use LoggerTrait;

    public function processPayment() {
        $this->log("Payment processed");
    }
}

// Использование
$user = new User();
$user->login(); // Выведет: LOG: User logged in

$payment = new PaymentSystem();
$payment->processPayment(); // Выведет: LOG: Payment processed
```

При использовании наследования, пришлось бы создавать общий базовый класс для User и PaymentSystem, но это совершенно разные классы.

Пространства имен и автозагрузка классов

Модульная структура программ

Включение файлов

- `include`
- `include_once`
- `require`
- **`require_once`**
 - если файла нет, то будет ошибка
 - Если файл уже был загружен ранее, он не выполнится повторно, но его код будет доступен

Магическая константа `__DIR__` - каталог, в котором находится запущенный скрипт.

Включение файлов

- При включении файла он полностью выполняется.
- Экспортировать только определенные функции из подключаемого файла нельзя.
- В подключаемом файле не должно быть исполняемого кода, только определения функций.
- Имена переменных, констант, функций, классов в разных подключаемых файлах должны быть уникальными

Проблема именования

Код всех файлов проекта находится в одном пространстве, поэтому имена переменных, констант, функций и классов в разных файлах должны быть разными.

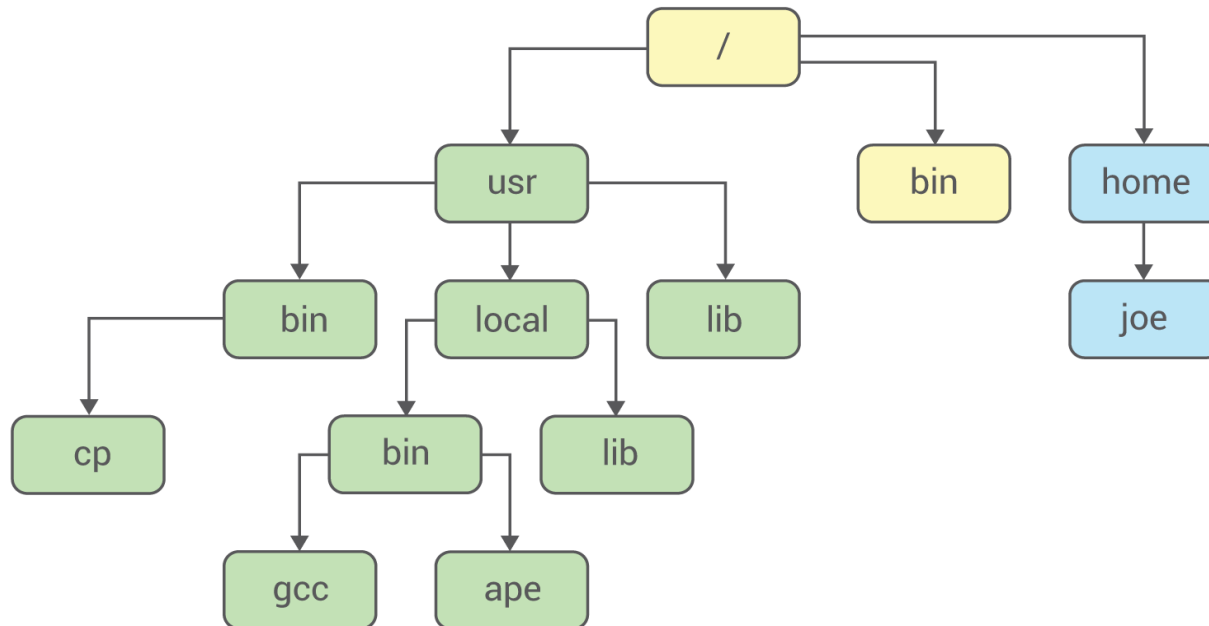
Можно добавлять длинные префиксы к отдельным названиям, но это неудобно.

Для изоляции кода используются ***пространства имен***. Это обладающие именем фрагменты программы, содержащий в себе классы, интерфейсы, функции, константы.

Пространства имен (namespaces)

Аналоги:

- Каталог в файловой системе - в разных каталогах файлы могут иметь одни и те же имена.
- Код города в телефонном номере
- Страны, города и улицы в почтовом адресе



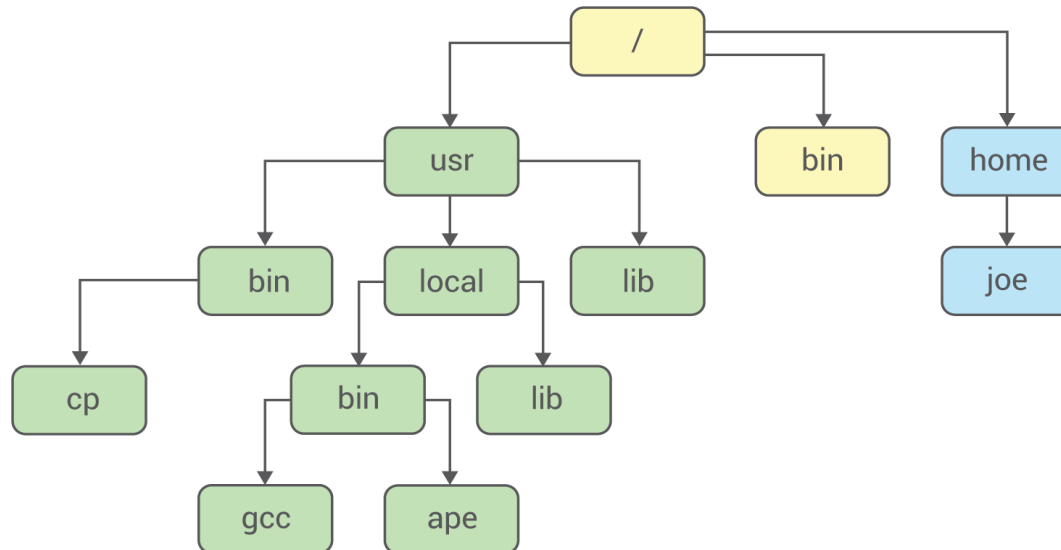
Пространства имен

Пространство имен - это обладающий именем фрагмент программы, содержащий в себе функции, переменные, константы и другие именованные элементы языка.

- Действие пространства имен распространяется только на классы, интерфейсы, функции и константы.
- Имя пространства имен не связано с именем файла.

Иерархия пространств имен в PHP

- Имя пространства имен не связано с именем файла.
- Глобальное пространство имен `\`, в котором классы, функции, константы создаются по умолчанию.
- Понятие текущего пространства имен
- Абсолютные и относительные пути к пространствам имен (аналогично путям в файловой системе)



Импорт сущностей из других пространств имен

Ключевое слово `use` позволяет импортировать функции, константы и классы из другого пространства имен

```
<?php
// Глобальное пространство имен
function sayHello() { echo "Global\n"; }
const APP_NAME = "Global App";

// Пространство MyNamespace
namespace MyNamespace;
function sayHello() { echo "MyNamespace\n"; }
const APP_NAME = "MyNamespace App";

// Импорт из глобального пространства
use function \sayHello as globalSayHello;
use const \APP_NAME as GLOBAL_APP_NAME;

globalSayHello(); // Global
echo GLOBAL_APP_NAME . "\n"; // Global App
sayHello(); // MyNamespace
echo APP_NAME . "\n"; // MyNamespace App
```

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`).