

ДОНСКОЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ

КАФЕДРА АВТОМАТИЗАЦИЯ ПРОИЗВОДСТВЕННЫХ ПРОЦЕССОВ

ЛЕКЦИЯ №4

Объектно-ориентированное
программирование в Python.
Инкапсуляция

СОСТАВИТЕЛЬ: КАНД. ТЕХН. НАУК БЫКАДОР В.С.

Общее представление об объектно-ориентированном программировании (ООП)

ООП является самой распространённой парадигмой в создании программного обеспечения. Главной особенностью ООП является возможность создавать программу в терминах объектов реального мира. То есть программа создается как информационная модель (с той или иной степенью точности) части реального мира. Программируя в рамках ООП, можно мыслить, что работа совершается как бы над реальными объектами, это значительно упрощает процесс разработки и поддержания программного обеспечения, особенно для крупных систем.

В классическом ООП («класс-ориентированном») любой объект должен относиться (быть экземпляром) какого-либо класса. Класс представляет собой набор:

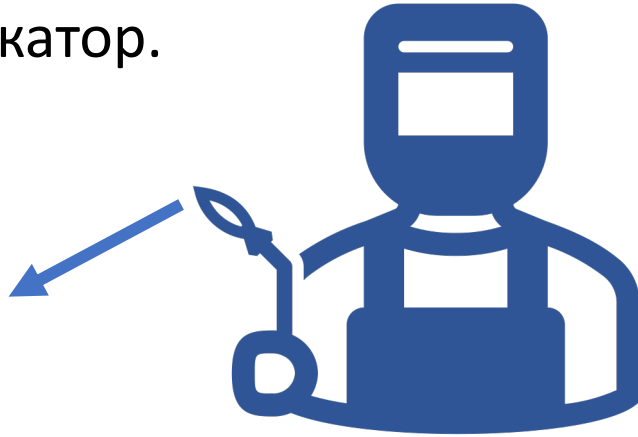
- 1) атрибутов характеризующих какую-либо сущность;
- 2) действий которые эта сущность может выполнять или над которой эти действия могут выполняться.

Сущность реального мира

Работник

Атрибуты:

1. Уникальный идентификатор.
2. Имя.
3. Отчество.
4. Фамилия.
5. Дата рождения.
6. Пол.
7. Отдел (цех).
8. Стаж работы.
9. Начисленная зарплата.



Операции:

1. Принять на работу.
2. Уволить с работы.
3. Перевести в подразделение.
4. Начислить зарплату.
5. Начислить надбавку.

Таким образом, мы получаем некоторое обобщенное описание какого-то произвольного работника и это обобщенное описание будет являться **классом**.

Далее, создавая конкретных работников, как экземпляры класса, мы будем заполнять для каждого работника соответствующие атрибуты. При этом действия, которые можно выполнять над работниками, останутся одними и теми же для любого работника, их не нужно будет как-то определять для каждого работника отдельно, тем самым достигает значительная «экономия кода», то есть однократно написанный код можно потом многократно использовать. Механизмы объектно-ориентированного программирования позаботятся о правильном вызове соответствующих частей кода.

Экземпляры класса

Работник №1

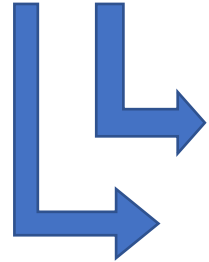
Работник №2

Работник №3



Термины

Атрибуты



Поля (закрытые атрибуты)

Свойства (открытые/публичные атрибуты)

Операции



Методы

Класс



Тип

Тип стандартной библиотеки наз. встроенными типом (классом).

Тип встроенной библиотеки наз. типом (классом) этой встроенной библиотеки.

Тип написанный самим программистом наз. пользовательским типом (классом).

Определение элементарного класса на языке Python

class

обязательное ключевое слово, которое обозначает, что код дальше будет относиться к классу.

Рабочий код класса: поля, свойства, конструктор, методы.

!СМЕЩЕНИЕ КОДА!

```
1 class Worker:
2     pass
```

•

Обязательное двоеточие в конце сигнатуры объявления класса, указывающее на начало блока кода класса.

Этот класс ничего не делает, но синтаксически он записан верно.

Конструктор класса в Python

Конструктор класса – это специальный метод, который вызывается при создании экземпляра класса.

Семантически конструктор представляется как функция и вообще методы в классах записываются как функции. У конструктора есть определенное название, которое зарезервировано и интерпретатор языка программирования Python понимает, что он имеет дело именно с конструктором класса.

Сигнатура конструктора класса в Python

`__init__`

Это специальное зарезервированное слово, указывающее на то, что этот метод именно конструктор класса.

`def __init__(self):`

`--`

Это два подряд идущих подчёркивания.

`--`

Это два подряд идущих подчёркивания.

В языке Python существует целый ряд зарезервированных методов с двумя нижними подчёркиваниями перед и после смыслового названия метода.

Эти методы предназначены для специальных целей и называются

дандер-методы или «**магические**» методы.

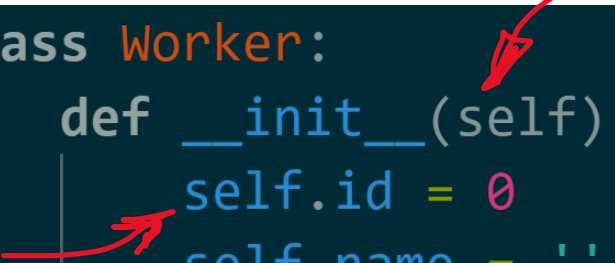
Конструктор класса «Worker» в Python

```
1  import datetime
2
3  class Worker:
4      def __init__(self):
5          self.id = 0
6          self.name = ''
7          self.middle_name = ''
8          self.last_name = ''
9          self.birth_date = datetime.datetime(1900)
10         self.gender = None
11         self.department = None
12         self.seniority = None
13         self.salary = 0
14
```

Создаем и
инициализируем в
конструкторе поля
класса Worker.

Что такое `self`?

```
class Worker:
    def __init__(self):
        self.id = 0
        self.name = ''
```



`self` – это указатель на конкретный экземпляр класса.

Как уже отмечалось ранее, класс представляет собой общее описание объекта, в данном случае это какой-то «рабочий». Но конкретные «рабочие» все разные, у них свои имена, фамилии, года рождения, зарплаты и так далее. А класс один, общий...

Так вот, чтобы у каждого конкретного «рабочего» была структура полей определенная в классе, а значения этих полей были свои собственные нужно иметь соответствующую «метку» вот этот самый **`self`**. Технически это означает, что в ОЗУ ЭВМ выделяется область памяти, для каждого конкретного «рабочего», в которой создаётся структура полей типа, а так как область памяти отдельная, то данные в эту структуру полей для каждого «рабочего» можно записывать свои собственные.

Почему интерпретатор Python сам не задаст ключевое слово `self`?

В общем случае, в классе могут быть поля, которые разделяются всеми его экземплярами, поэтому, когда программист явно прописывает ключевое слово `self`, то программист явно указывает интерпретатору Python, что нужно для каждого экземпляра сделать копию структуры полей и записывать данные для каждого экземпляра отдельно.

Пример разделяемого и собственного значения поля



Создание экземпляра класса «Worker»

```
import datetime

from clsWorker import Worker

worker01 = Worker()

worker01.id = 123343

worker01.name = 'Пётр'

worker01.middle_name = 'Семёнович'

worker01.last_name = 'Иванов'

worker01.birth_date = datetime.datetime(1985, 3, 10)

worker01.gender = 'male'

worker01.department = 24

worker01.seniority = 'senior'

worker01.salary = 150000
```

```
worker02 = Worker()

worker02.id = 101000

worker02.name = 'Ира'

worker02.middle_name = 'Ивановна'

worker02.last_name = 'Сидорова'

worker02.birth_date = datetime.datetime(1998, 3, 10)

worker02.gender = 'female'

worker02.department = 24

worker02.seniority = 'junior'

worker02.salary = 50000
```

Взаимодействие с экземплярами класса «Worker»

```
print(worker01.last_name)  
print(worker02.last_name)
```



```
/букV1/Yande  
Иванов  
Сидорова  
РБ-С-Н-Н-Н
```

ИНКАПСУЛЯЦИЯ ДАННЫХ В КЛАССЕ

Инкапсуляция является важнейшим принципом ООП и предполагает, что данные экземпляра класса скрыты от внешней части программы. Это важно, так как экземпляр класса сам владеет и сам модифицирует свои данные, тем самым обеспечивая их целостность и скрывает внутреннюю организацию данных от внешней части программы.

Для реализации принципа инкапсуляции данных поля класса объявляются частными, то есть доступ к этим полям будет иметь только данный экземпляр класса внутри себя. В разных языках программирования по-разному синтаксически объявляются частные поля, в Python необходимо перед именем поля поставить два подчёркивания и тогда это поле будет частным (пример на следующем слайде).

СИНТАКСИС ЧАСТНЫХ ПОЛЕЙ

clsWorker.py > Worker > __init__

```
1  import datetime
2
3  class Worker:
4      def __init__(self):
5          self.__id = 0
6          self.__name = ''
7          self.__middle_name = ''
8          self.__last_name = ''
9          self.__birth_date = datetime.datetime(1900, 1, 1)
10         self.__gender = None
11         self.__department = None
12         self.__seniority = None
13         self.__salary = 0
```

Это два подчёркивания.

Теперь если мы обратимся к любому полю экземпляра класса, то получим ошибку, так из внешней части программы не будет ничего «видно» что там «внутри» экземпляра класса.

КАК ЗАДАВАТЬ ЗНАЧЕНИЯ ЧАСТНЫХ ПОЛЕЙ ИЗ ВНЕШНЕЙ ПРОГРАММЫ

Может возникнуть вопрос: «А как тогда присваивать значения полям класса?». Присваивать значения полям можно через методы, которые специально работают с полями – эти методы часто называют **методами доступа**.

Но также можно использовать и **конструктор класса**, который позволяет сразу задать значения частных полей. Задание значений полей через конструктор используют специально тогда, когда разработчик класса желает гарантировать то, что нужные поля будут точно заданы пользователем класса, так как в противном случае экземпляр класса не создается и программа не выполнится. Можно задать все поля через конструктор, можно не все – это всё определяется конкретным классом.

ЗАДАВАТЬ ЗНАЧЕНИЯ ЧАСТНЫХ ПОЛЕЙ ЧЕРЕЗ КОНСТРУКТОР

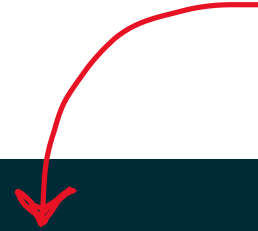
В нашем учебном классе давайте через конструктор зададим все те поля которые однозначно относятся к работнику:

- 1) Идентификатор
- 2) Имя
- 3) Отчество
- 4) Фамилия
- 5) Дата рождения
- 6) Пол



```
2  class Worker:
3      def __init__(self, id, name, middle_name, last_name, birth_date, gender):
4          self.__id = id
5          self.__name = name
6          self.__middle_name = middle_name
7          self.__last_name = last_name
8          self.__birth_date = birth_date
9          self.__gender = gender
10
11         self.__department = None
12         self.__seniority = None
13         self.__salary = 0
14
```

Объявляем параметры конструктора как у функции.



ИНИЦИАЛИЗАЦИЯ ПОЛЕЙ КЛАССА ПРИ ЕГО СОЗДАНИИ

```
1  import datetime
2  from clsWorker import Worker
3
4  birth_date01 = datetime.datetime(1985, 3, 10)
5  worker01 = Worker(123343, 'Пётр', 'Семёнович', 'Иванов', birth_date01, 'male')
6
7  worker02 = Worker(
8      id=101000,
9      name='Ира',
10     last_name='Сидорова',
11     middle_name = 'Ивановна',
12     gender = 'female',
13     birth_date = datetime.datetime(1998, 10, 15)
14 )
15
```

Позиционный способ задания параметров

Именованный способ задания параметров

КАК ПОЛУЧИТЬ ДОСТУП К ЧАСТНОМУ ПОЛЮ КЛАССА

```
7 worker02 = Worker(  
8     id=101000,  
9     name='Ира',  
10    last_name='Сидорова',  
11    middle_name = 'Ивановна',  
12    gender = 'female',  
13    birth_date = datetime.datetime(1998, 10, 15)  
14 )  
15  
16 print(worker02.__last_name)  
17
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests> & C:/Users  
/Programs/Python/Python310/python.exe "c:/Users/bykvi/YandexDisk/В работ  
ests/main.py"
```

```
Traceback (most recent call last):
```

```
File "c:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests\main.py",  
line 16, in <module>
```

```
    print(worker02.__last_name)
```

```
AttributeError: 'Worker' object has no attribute '__last_name'
```

```
PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests>
```

ПУБЛИЧНЫЕ МЕТОДЫ ДОСТУПА КЛАССА

А если нужно обратиться к частному полю класса то, как быть? В этом случае нужен метод доступа, который будет публичным, то есть к нему можно будет выполнить обращение из вне класса, но сам метод доступа будет работать с частным полем, то есть будет являться некоторым посредником между внешней частью программы и частными (закрытыми) полями экземпляра класса. Например, для поля **__last_name** мы можем определить следующий метод доступа для получения его значения — метод

get_last_name ().

СОЗДАНИЕ ПУБЛИЧНОГО МЕТОДА ДОСТУПА

```
3 class Worker:
4     def __init__(self, id, name, middle_name,
5         self.__id = id
6         self.__name = name
7         self.__middle_name = middle_name
8         self.__last_name = last_name
9         self.__birth_date = birth_date
10        self.__gender = gender
11
12        self.__department = None
13        self.__seniority = None
14        self.__salary = 0
15
16    def get_last_name(self):
17        return self.__last_name
18
```

ПОЛУЧИТЬ ДАННЫЕ ЧАСТНОГО ПОЛЯ ЧЕРЕЗ МЕТОД ДОСТУПА

```
7  worker02 = Worker(  
8      id=101000,  
9      name='Ира',  
10     last_name='Сидорова',  
11     middle_name = 'Ивановна',  
12     gender = 'female',  
13     birth_date = datetime.datetime(1998, 10, 15)  
14 )  
15  
16 print(worker02.get_last_name())  
17
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests> & C:/Users/b  
/Programs/Python/Python310/python.exe "c:/Users/bykvi/YandexDisk/В работе/  
ests/main.py"
```

Сидорова

```
PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests> █
```


ЗАПИСАТЬ ДАННЫЕ ЧАСТНОГО ПОЛЯ ЧЕРЕЗ МЕТОД ДОСТУПА

```
3 class Worker:
4     def __init__(self, id, name, middle_name,
5         self.__id = id
6         self.__name = name
7         self.__middle_name = middle_name
8         self.__last_name = last_name
9         self.__birth_date = birth_date
10        self.__gender = gender
11
12        self.__department = None
13        self.__seniority = None
14        self.__salary = 0
15
16    def get_last_name(self):
17        return self.__last_name
18
19    def set_last_name(self, value):
20        self.__last_name = value
```

```
7 worker02 = Worker(
8     id=101000,
9     name='Ипа',
10    last_name='Сидорова',
11    middle_name = 'Ивановна',
12    gender = 'female',
13    birth_date = datetime.datetime(1998, 10, 15)
14 )
15
16 worker02.set_last_name('Петрова')
17 print(worker02.get_last_name())
18
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests> & C:/Users/
/Programs/Python/Python310/python.exe "c:/Users/bykvi/YandexDisk/В работе
ests/main.py"
```

Петрова

```
PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests> █
```

«ГЕТТОРЫ», «СЕТТОРЫ» И СВОЙСТВА

Методы доступа, которые позволяют получить данные из частного поля класса называются **«геттерами»** (get), а методы доступа которые позволяют записать данные в частное поле класса называются **«сеттерами»** (set).

Работа с частными поля классов через методы доступа является классическим подходом, но язык Python, как и многие другие современные языки программирования, поддерживающие объектно-ориентированное программирование, предоставляет так называемые **свойства**.

Свойства – это те же по сути **«гетторы»** и **«сетторы»**, но синтаксически оформленные немного по-другому, что позволяет пользователю класса работать немного удобнее с классом. Давайте перепишем предыдущий пример используя свойства.

СВОЙСТВА ДЛЯ ДОСТУПА К ПОЛЯМ КЛАССА

```
3 class Worker:
4     def __init__(self, id, name, middle_name,
5         self.__id = id
6         self.__name = name
7         self.__middle_name = middle_name
8         self.__last_name = last_name
9         self.__birth_date = birth_date
10        self.__gender = gender
11
12        self.__department = None
13        self.__seniority = None
14        self.__salary = 0
15
16        @property
17        def last_name(self):
18            return self.__last_name
19
20        @last_name.setter
21        def last_name(self, value):
22            self.__last_name = value
23
```

Свойства имеют синтаксис функций, при этом свойство для получения и записи данных имеет одно и тоже имя **last_name**.

Для того, чтобы PVM считала эти функции свойствами нужно добавить так называемые **декораторы** перед каждой функцией.

Декоратор, который делает функцию «геттером» записывается как

@property

Декоратор, который делает функцию «сеттером» записывается как

@имя_функции.setter

РАБОТА СО СВОЙСТВАМИ

Внешне работа со свойствами похожа на работу с публичными полями.

```
8     id=101000,  
9     name='Ира',  
10    last_name='Сидорова',  
11    middle_name = 'Ивановна',  
12    gender = 'female',  
13    birth_date = datetime.datetime(1998, 10, 15)  
14    )  
15  
16    print(worker02.last_name)  
17    worker02.last_name = 'Петрова'  
18    print(worker02.last_name)  
19
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

```
PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests> & C:/Users/I  
/Programs/Python/Python310/python.exe "c:/Users/bykvi/YandexDisk/В работе,  
ests/main.py"
```

```
Сидорова  
Петрова
```

```
PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests> 
```

ЗАЧЕМ ВООБЩЕ НУЖНЫ СВОЙСТВА

Может возникнуть вопрос, связанный с тем, зачем вводить абстракцию частных полей и публичных свойств, почему сразу не сделать поля публичными как это было сделано в самом начале рассмотрения примера с классом?

- 1) Частные поля могут требоваться для работы самого класса, то есть такие поля как переменные используются внутри класса и не предназначены для публикации, более того доступ к таким полям из вне может помешать нормальной работе класса.
- 2) Другой причиной разделение на поля и свойства является то, что у полей нет информации о том, как работать с теми данными, которые им присваиваются.

КАК БЫ БЫЛА РЕАЛИЗОВАНА ПРОВЕРКА ЗНАЧЕНИЙ ПУБЛИЧНОГО ПОЛЯ БЕЗ СВОЙСТВА

```
10 new_salary = 1000
11 if new_salary >= 13500:
12     worker01.salary = new_salary
13 else:
14     print('Ошибка!')
15
16 new_salary = 25000
17 if new_salary >= 13500:
18     worker02.salary = new_salary
19 else:
20     print('Ошибка!')
```

```
10 def salary(salary, worker):
11     if salary >= 13500:
12         worker.salary = salary
13     else:
14         print('Ошибка!')
15
16
17 salary(1000, worker01)
18 salary(25000, worker02)
19
```

Проверка зарплаты на то, не меньше ли она минимально допустимой.

Проверка выполняется во внешнем коде, который использует экземпляр класса.

Минус такого решения связан с рефакторингом, расширением и переносом кода.

МИНУСЫ ПРОВЕРКИ ЗНАЧЕНИЙ ПУБЛИЧНОГО ПОЛЯ БЕЗ СВОЙСТВА

Минусы вышеприведенного решения связаны с рефакторингом, расширением, масштабированием и переносом кода.

Эти проблемы возникают, потому что публичное поле ничего не знает о тех данных, которые ему присваиваются и нет возможности как-то в этом поле написать правила обработки данных этого же поля. Такие правила часто называют **«бизнес-логика»** или **«бизнес-правила»**.

И вот здесь как раз может помочь концепция разделения на частные поля и публичные свойства. Поля будут данные хранить, а свойства реализовывать всякие проверки (бизнес-логику) перед тем как эти данные будут записаны в поле из вне или считаны из поля для внешнего потребителя.

Поэтому давайте обратно сделаем рассматриваемое поле частным, то есть `__salary`, добавим соответствующие свойства (или методы доступа), в которых пропишем нужную «бизнес-логику» работы с этим полем.

ПРОВЕРКА ЗНАЧЕНИЙ ЧАСТНОГО ПОЛЯ ПУБЛИЧНЫМ СВОЙСТВОМ

```
12 self.__department = None
13 self.__seniority = None
14 self.__salary = 13500
15
16 @property
17 def salary(self):
18     return self.__salary
19
20 @salary.setter
21 def salary(self, value):
22     if value >= 13500:
23         self.__salary = value
24     else:
25         print('Ошибка!')
```

```
12
13 worker01.salary = 1000
14 worker02.salary = 25000
15 print(worker02.salary)
```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE

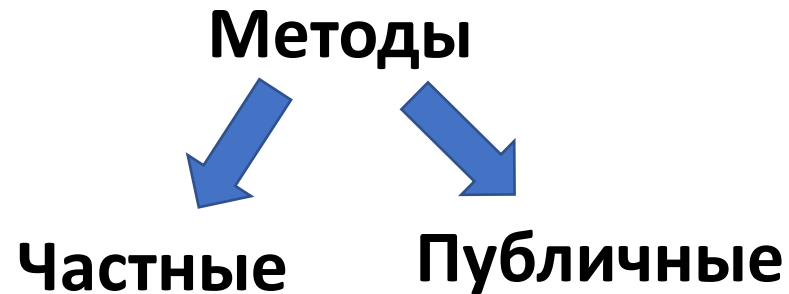
```
PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests> & C:/Programs/Python/Python310/python.exe "c:/Users/bykvi/YandexDisk/B
ests/main.py"
```

Ошибка!
25000

```
PS C:\Users\bykvi\YandexDisk\В работе\Лекции по Python\tests> █
```


МЕТОДЫ

Методы в классах позволяют выполнять преобразование данных полей класса. Как правило методами моделируют те действия которые может выполнять объект или те действия которые могут выполняться над объектом. Но не следует последнее принимать как руководство, так как методы могут выражать и абстрактные действия, которые у реального объекта отсутствуют, но необходимые с точки зрения концепции разрабатываемой системы классов.



Частные методы доступны для вызова только внутри того класса в котором они определены.

Публичные методы доступны для вызова как внутри того класса в котором они определены, так и извне.

РЕАЛИЗАЦИЯ МЕТОДОВ

Теперь давайте реализуем единственный метод «Начислить надбавку».

Методы семантически представляют собой функции с обязательным параметром `self`, который указывает на экземпляр класса. В отличие от свойств методу можно передавать параметры.

```
19     @property
20     def salary(self):
21         return self.__salary
22
23     @salary.setter
24     def salary(self, value):
25         if value >= 13500:
26             self.__salary = value
27         else:
28             print('Ошибка!')
29
30     def add_to_salary(self, procent):
31         self.salary += (procent / 100) * self.salary
32
```

ВЫЗОВ МЕТОДОВ

```
13 worker01.salary = 15000
14 worker02.salary = 25000
15
16 print('')
17 print('Зарплата:')
18 print(worker01.salary)
19 print(worker02.salary)
20
21 worker01.add_to_salary(10)
22 worker02.add_to_salary(12)
23
24 print('')
25 print('-----')
26 print('Зарплата после надбавки:')
27 print(worker01.salary)
28 print(worker02.salary)
```

Зарплата:

15000

25000

Зарплата после надбавки:

16500.0

28000.0

ИНТЕРФЕЙС КЛАССА

В интерфейс класса входят:

- 1) конструктор;**
- 2) публичные свойства;**
- 3) публичные методы.**