Объектно-ориентированное программирование. Язык Python

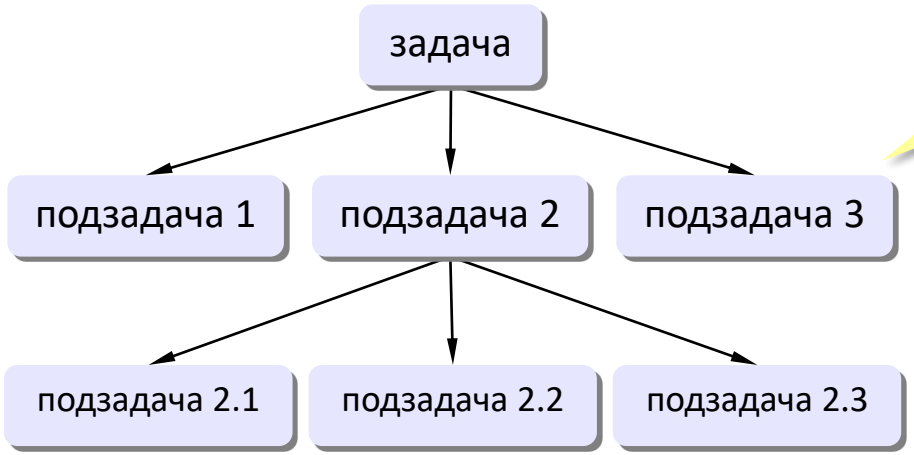
Зачем нужно что-то новое?

Главная проблема – **сложность**!

- программы из миллионов строк
- тысячи переменных и массивов

Э. Дейкстра: «Человечество еще в древности придумало способ управления сложными системами: «**разделяй и властвуй**»».

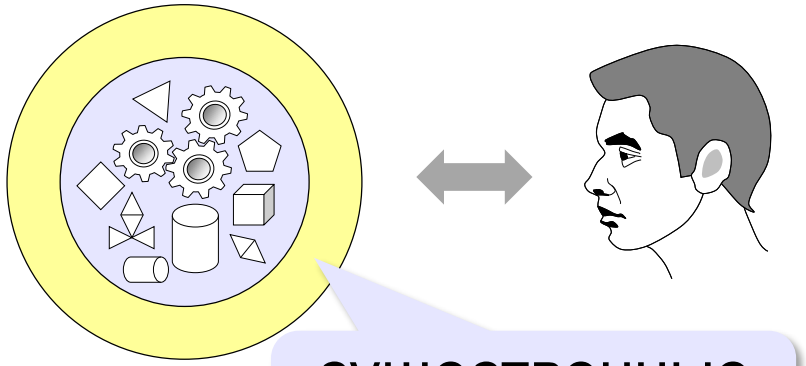
Структурное программирование:



декомпозиция по задачам

человек мыслит иначе, объектами

Как мы воспринимаем объекты?



существенные свойства

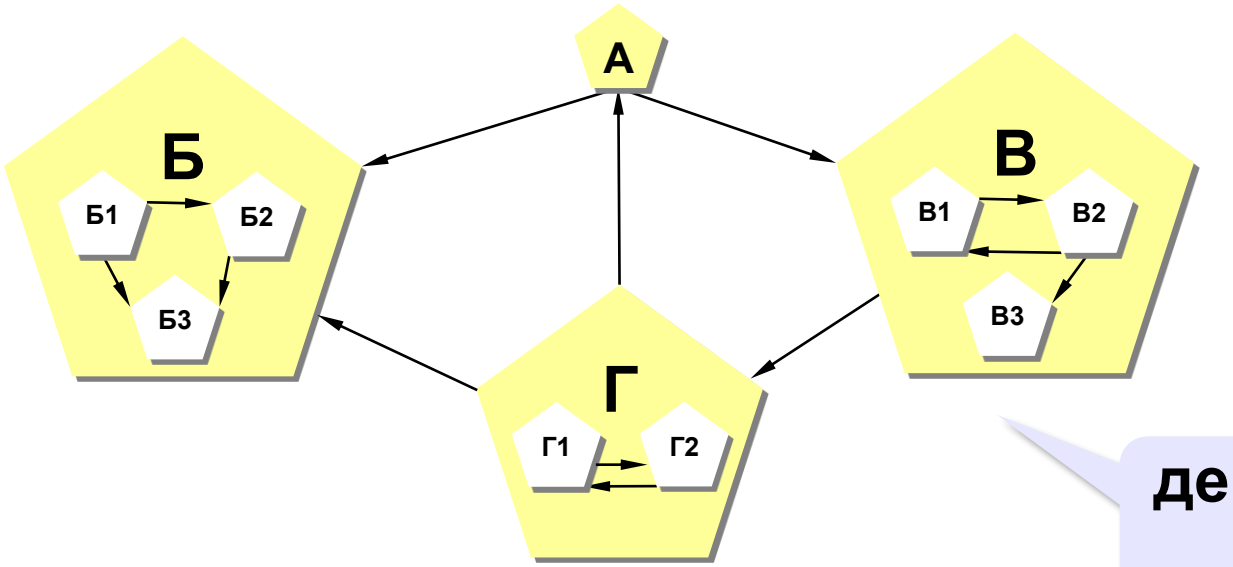
Абстракция – это выделение существенных свойств объекта, отличающих его от других объектов.

Разные цели – **разные модели**!

Использование объектов

Программа – множество объектов (моделей), каждый из которых обладает своими свойствами и поведением, но его внутреннее устройство скрыто от других объектов.

Нужно «разделить» задачу на объекты!



декомпозиция по объектам

С чего начать?

Объектно-ориентированный анализ (ООА):

- выделить **объекты**
- определить их существенные **свойства**
- описать **поведение** (команды, которые они могут выполнять)

Что такое объект?

Объектом можно назвать то, что имеет чёткие границы и обладает *состоянием* и *поведением*.

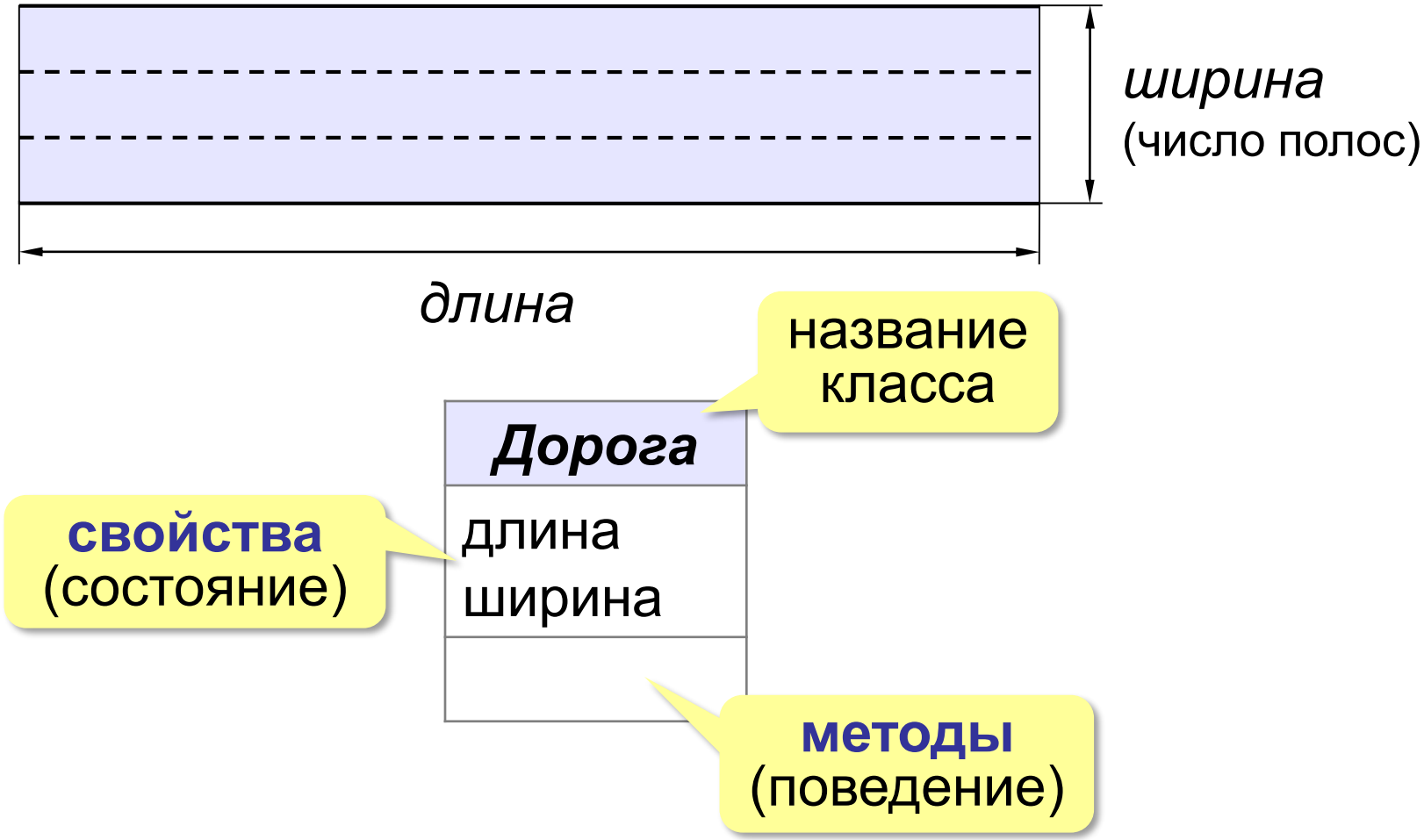
Состояние определяет поведение:

- лежачий человек не прыгнет
- незаряженное ружье не выстрелит

Класс – это множество объектов, имеющих общую структуру и общее поведение.

Модель дороги с автомобилями

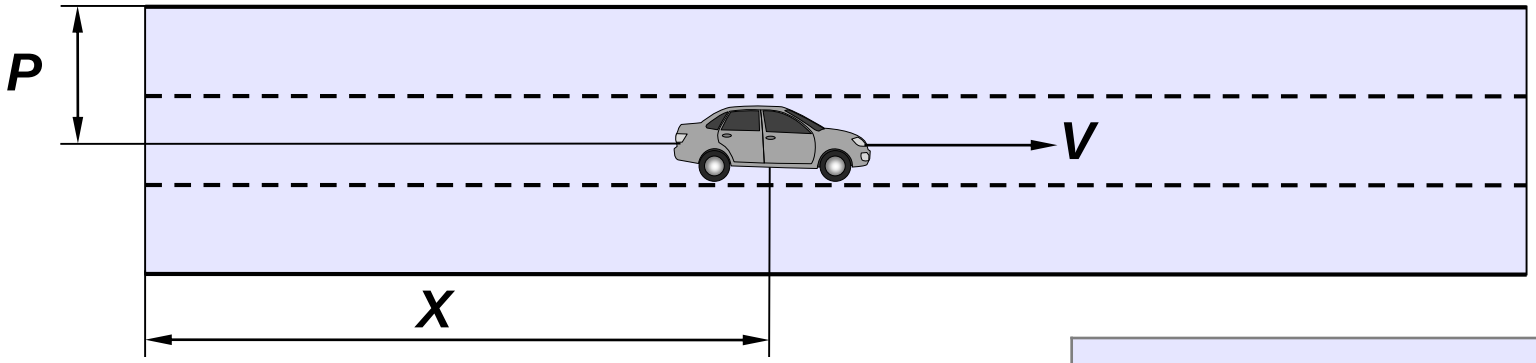
Объект «Дорога»:



Модель дороги с автомобилями

Объект «Машина»:

свойства: координаты и скорость



- все машины одинаковы
- скорость постоянна
- на каждой полосе – одна машина
- если машина выходит за правую границу дороги, вместо нее слева появляется новая машина

Машина
X (координата) P (полоса) V (скорость)
двигаться

Метод – это процедура или функция, принадлежащая классу объектов.

Модель дороги с автомобилями

Взаимодействие объектов:

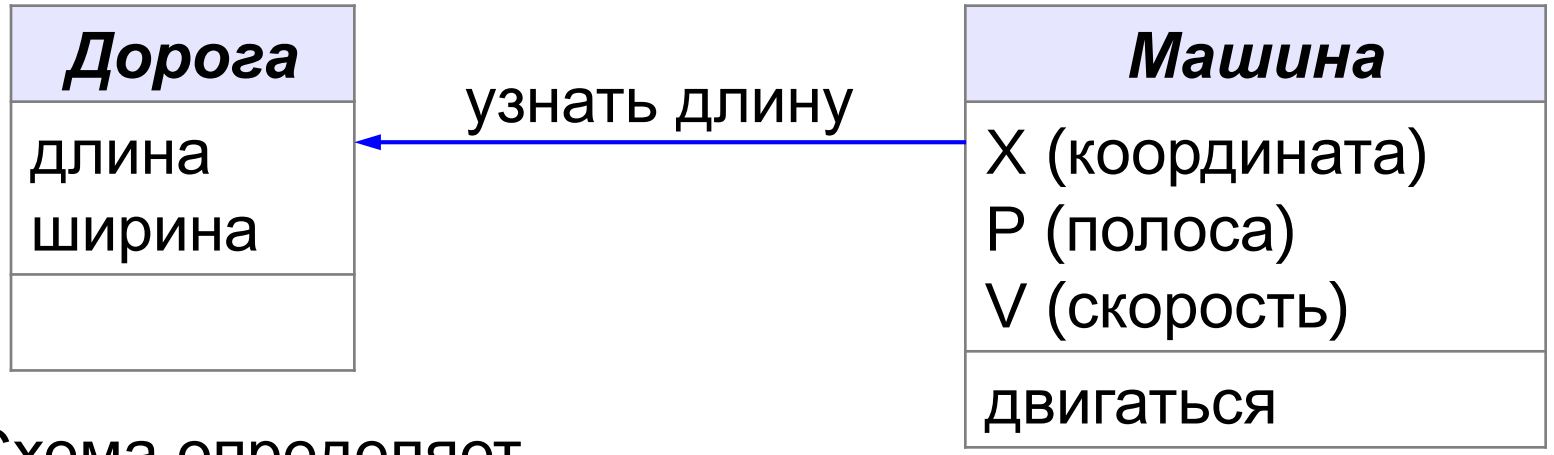


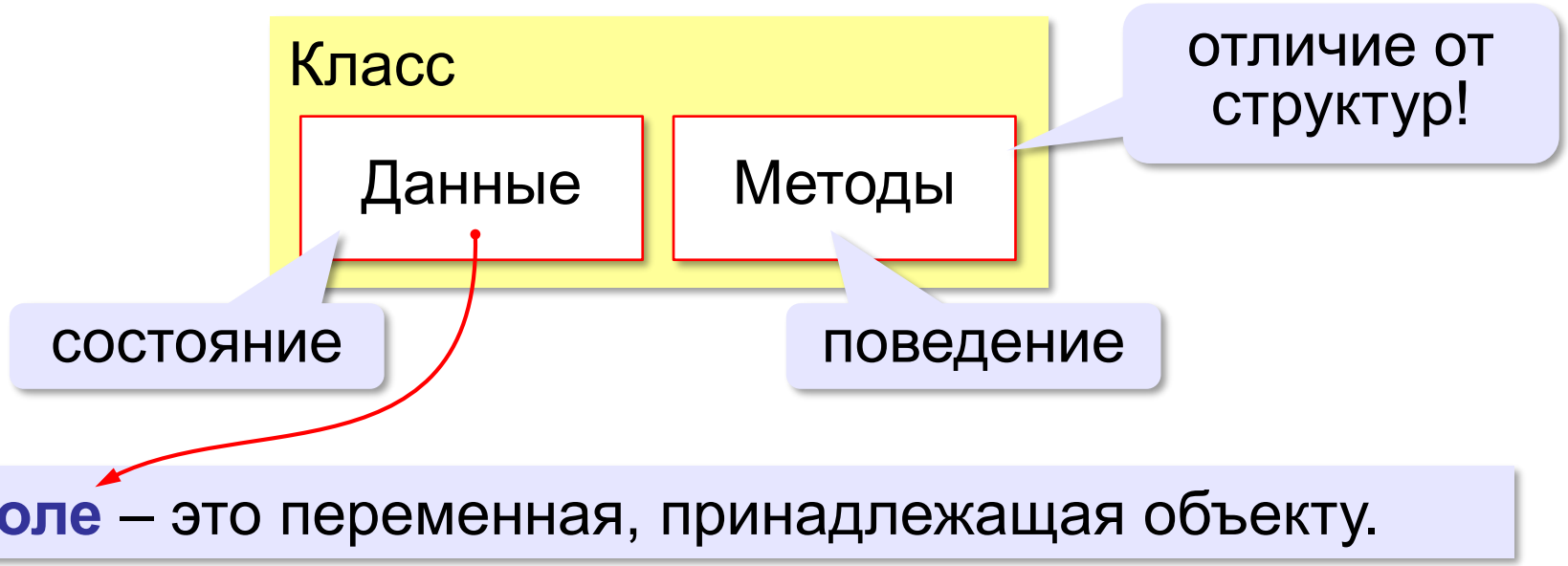
Схема определяет

- **свойства** объектов
- **методы**: операции, которые они могут выполнять
- **связи** (обмен данными) между объектами

Ни слова о внутреннем устройстве объектов!

Классы

- программа – множество взаимодействующих **объектов**
- любой объект – экземпляр какого-то **класса**
- **класс** – описание группы объектов с общей структурой и поведением



Поле – это переменная, принадлежащая объекту.

Класс «Дорога»

Описание класса:

```
class TRoad:
  pass
```

Объекты-экземпляры не создаются!

Создание объекта:

```
road = TRoad()
```

вызов конструктора

Конструктор – это метод класса, который вызывается для создания объекта этого класса.

Конструктор по умолчанию строится автоматически!

Новый конструктор – добавлений полей

initialization – начальные установки

```
class TRoad:
  def __init__( self ):
    self.length = 0
    self.width = 0
```

ссылка для обращения к самому объекту

оба поля обнуляются

точечная запись

Конструктор задаёт начальные значения полей!

```
road = TRoad()
road.length = 60
road.width = 3
```

изменение значений полей

Конструктор с параметрами

автоматически

```
class TRoad:
  def __init__( self, length0, width0 ):
    self.length = length0
    self.width = width0
```

Вызов:

```
road = TRoad( 60, 3 )
```

Нет защиты от неверных входных данных!

Защита от неверных данных

```
class TRoad:
  def __init__( self, length0, width0 ):
    if length0 > 0:
      self.length = length0
    else:
      self.length = 0
    if width0 > 0:
      self.width = width0
    else:
      self.width = 0
```

```
self.length = length0 if length0 > 0 else 0
self.width = width0 if width0 > 0 else 0
```

Класс «Машина»

```
class TCar:
  def __init__( self, road0, p0, v0 ):
    self.road=road0
    self.P=p0
    self.V=v0
    self.X=0
```

дорога, по которой едет

полоса

скорость

координата

Класс «Машина» – метод move

```
class TCar:
  def __init__( self, road0, p0, v0 ):
    ...
  def move( self ):
    self.X += self.V
    if self.X > self.road.length:
      self.X = 0
```

перемещение за Δt = 1

если за пределами дороги

Равномерное движение:

$$X = X_0 + V \cdot \Delta t$$

$\Delta t = 1$ интервал дискретизации

перемещение за одну единицу времени

Основная программа

```
N = 3
cars = []
for i in range(N):
  cars.append( TCar(road, i+1, 2*(i+1)) )

for k in range(100):   # 100 шагов
  for i in range(N):   # для каждой машины
    cars[i].move()

print( "После 100 шагов:" )
for i in range(N):
  print( cars[i].X )
```

Что в этом хорошего и плохого?

ООП – это метод разработки **больших** программ!

- основная программа – простая и понятная
- классы могут разрабатывать разные программисты независимо друг от друга (+интерфейс!)
- повторное использование классов

- неэффективно для небольших задач

Зачем скрывать внутреннее устройство?

Объектная модель задачи:

- защита внутренних данных
- проверка входных данных на корректность
- изменение устройства с сохранением интерфейса

Инкапсуляция («помещение в капсулу») – скрытие внутреннего устройства объектов.

Также объединение данных и методов в одном объекте!

Пример: класс «перо»

```
class TPen:
  def __init__( self ):
    self.color = "000000"
```

R G B

По умолчанию все члены класса открытые (в других языках – **public**)!

```
class TPen:
  def __init__( self ):
    self.__color = "000000"
```

Как обращаться к полю?

Имена скрытых полей (**private**) начинаются с двух знаков подчёркивания!

Пример: класс «перо»

```
class TPen:
  def __init__( self ):
    self.__color="000000"
  def getColor( self ):
    return self.__color
  def setColor( self, newColor ):
    if len(newColor) !=6:
      self.__color="000000"
    else:
      self.__color=newColor
```

метод чтения

метод записи

если ошибка, чёрный цвет

Защита от неверных данных!

Пример: класс «перо»

Использование:

```
pen = TPen()
pen.setColor ( "FFFF00" )
print ( "цвет пера:", pen.getColor() )
```

установить цвет

прочитать цвет

Не очень удобно!

```
pen.color = "FFFF00"
print ( "цвет пера:", pen.color )
```

Свойство color

Свойство – это способ доступа к внутреннему состоянию объекта, имитирующий обращение к его внутренней переменной.

```
class TPen:
  def __init__ ( self ):
    ...
  def __getColor ( self ):
    ...
  def __setColor ( self, newColor ):
    ...
  color = property ( __getColor,
                     __setColor )
```

метод чтения

свойство

метод записи

```
pen.color = "FFFF00"
print ( "цвет пера:", pen.color )
```

Изменение внутреннего устройства

Удобнее хранить цвет в виде числа:

```
class TPen:
  def __init__( self ):
    self.__color = 0
  def __getColor( self ):
    return "{:06x}".format( self.__color )
  def __setColor( self, newColor ):
    if len(newColor) != 6:
      self.__color = 0
    else:
      self.__color = int( newColor, 16 )
  color = property (__getColor, __setColor)
```

число

число

число

Интерфейс не изменился!

Преобразование int → hex

Целое – в шестнадцатеричную запись:

16711935 → "FF00FF"

```
x = 16711935
sHex = "{:x}".format(x)
```

в шестнадцатеричной системе

? Что плохо?

255 → "FF" "0000FF"

правильно так!

```
x = 16711935
sHex = "{:06x}".format(x)
```

дополнить нулями слева

занять 6 позиций

Преобразование hex → int

"FF00FF" → 16711935

```
sHex = "FF00FF"
x = int ( sHex, 16 )
```

система счисления

Свойство «только для чтения»

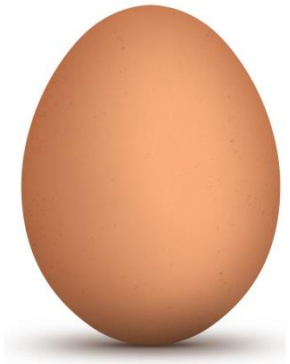
Скорость машины можно только читать:

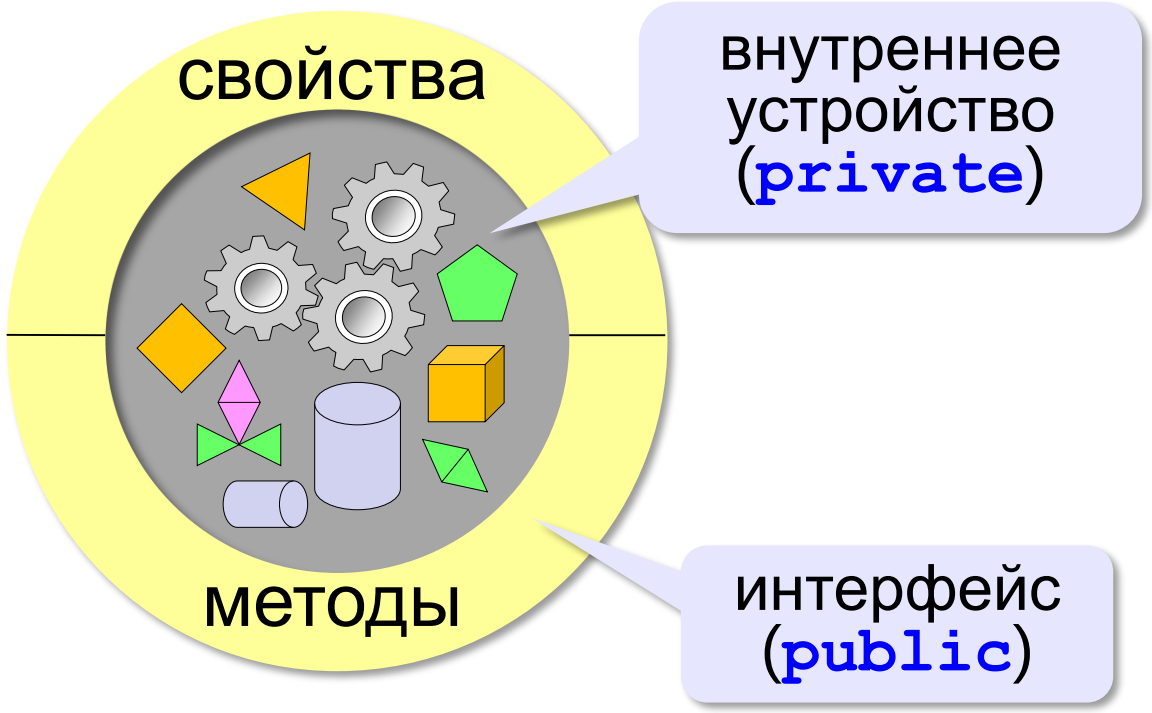
```
class TCar:
  def __init__( self ):
    self.__v=0
  v=property( lambda x: x.__v )
```

нет метода записи

Скрытие внутреннего устройства

Инкапсуляция («помещение в капсулу»)



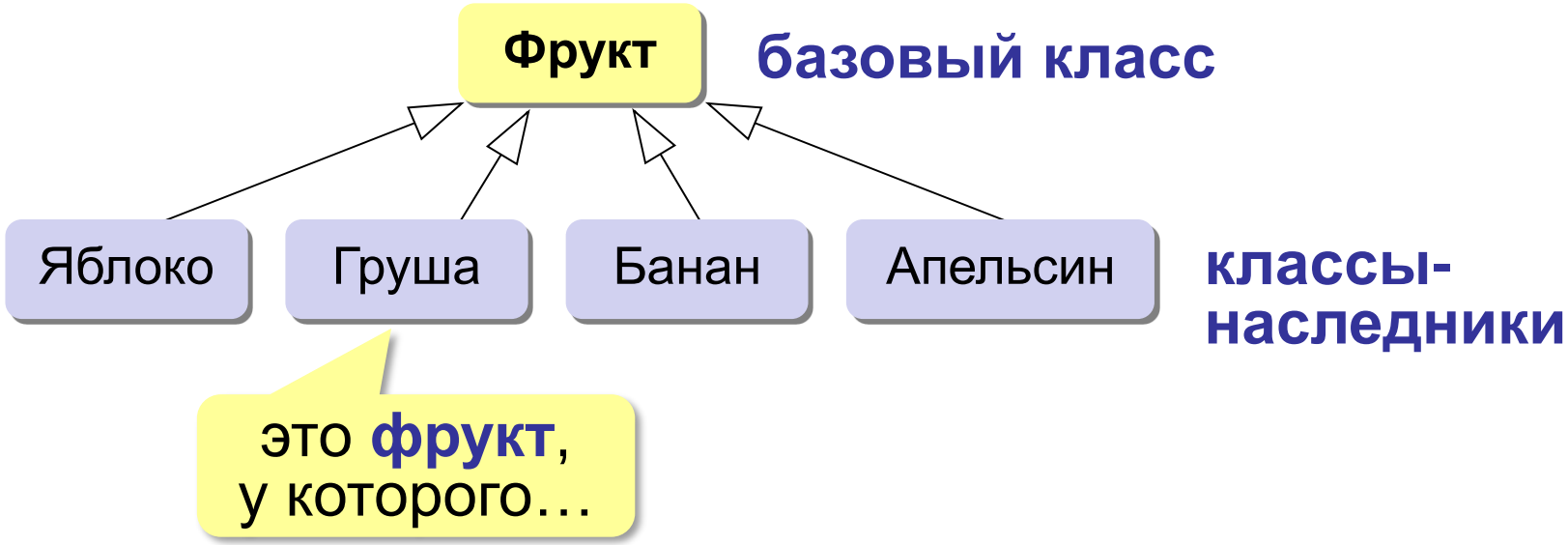


Классификации

? Что такое классификация?

Классификация – разделение изучаемых объектов на группы (классы), объединенные общими признаками.

? Зачем это нужно?



Что такое наследование?

класс *Двудольные*
семейство *Бобовые*
род *Клевер*
горный клевер

наследует свойства (имеет все свойства)

Класс Б является **наследником** класса А, если можно сказать, что Б – **это разновидность** А.

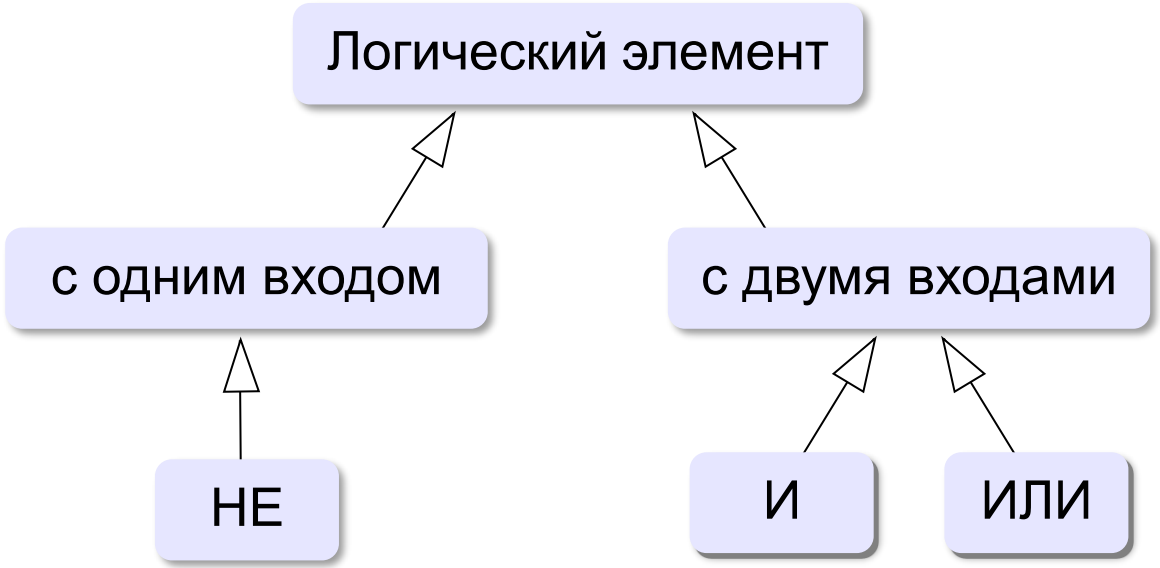
✓ яблоко – фрукт — яблоко – **это** фрукт

✓ горный клевер – клевер — горный клевер – **это** растение рода *Клевер*

✗ машина – двигатель

машина **содержит** двигатель (часть – целое)

Иерархия логических элементов



Объектно-ориентированное программирование – это такой подход к программированию, при котором программа представляет собой множество взаимодействующих **объектов**, каждый из которых является экземпляром определенного **класса**, а классы образуют иерархию **наследования**.

Базовый класс

ЛогЭлемент
In1 (вход 1) In2 (вход 2) Res (результат)
calc

```
class TLogElement:
  def __init__ ( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False
```

поле доступно наследникам!

? Зачем хранить результат?

можно моделировать элементы с памятью (триггеры)

Базовый класс

```
class TLogElement:
  def __init__( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False
  def __setIn1 ( self, newIn1 ):
    self.__in1 = newIn1
    self.calc()
  def __setIn2 ( self, newIn2 ):
    self.__in2 = newIn2
    self.calc()
  In1 = property (lambda x: x.__in1, __setIn1)
  In2 = property (lambda x: x.__in2, __setIn2)
  Res = property (lambda x: x._res )
```

пересчёт выхода

только для чтения

Метод `calc`

Как написать метод `calc`?

```
class TLogElement:
    ...
    def calc( self ):
        pass
```

заглушка

Нужно запретить создавать объекты **`TLogElement`**!

Абстрактный класс

- все логические элементы должны иметь метод `calc`
- метод `calc` невозможно написать, пока неизвестен тип логического элемента

Абстрактный метод – это метод класса, который объявляется, но не реализуется в классе.

Абстрактный класс – это класс, содержащий хотя бы один абстрактный метод.

нет логического элемента «вообще», как не «фрукта вообще», есть конкретные виды

Нельзя создать объект абстрактного класса!

`TLogElement` – абстрактный класс из-за метода `calc`

Абстрактный класс

```
class TLogElement:
  def __init__ ( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False

    if not hasattr ( self, "calc" ):
      raise NotImplementedError(
            "Нельзя создать такой объект!")
```

если у объекта нет атрибута (поля или метода) с именем `calc`...

создать («поднять», «выбросить») исключение

Что такое полиморфизм?

```
class TLogElement:
  def __init__( self ):
    ...
  def __setIn1 ( self, newIn1 ):
    self.__in1 = newIn1
    self.calc()
```

для каждого наследника вызывается свой метод **calc**

Полиморфизм – это возможность классов-наследников по-разному реализовать метод с одним и тем же именем.

греч.: $\pi o \lambda \upsilon$ — много, $\mu o \rho \varphi \eta$ — форма