Объектно-ориентированное программирование на Python

Осипов Никита Алексеевич

ЛЕКЦИЯ 3. ИНКАПСУЛЯЦИЯ

Учебные вопросы:

1. Понятие инкапсуляции.
2. Использование специальных методов доступа.
3. Применение аннотаций свойств.

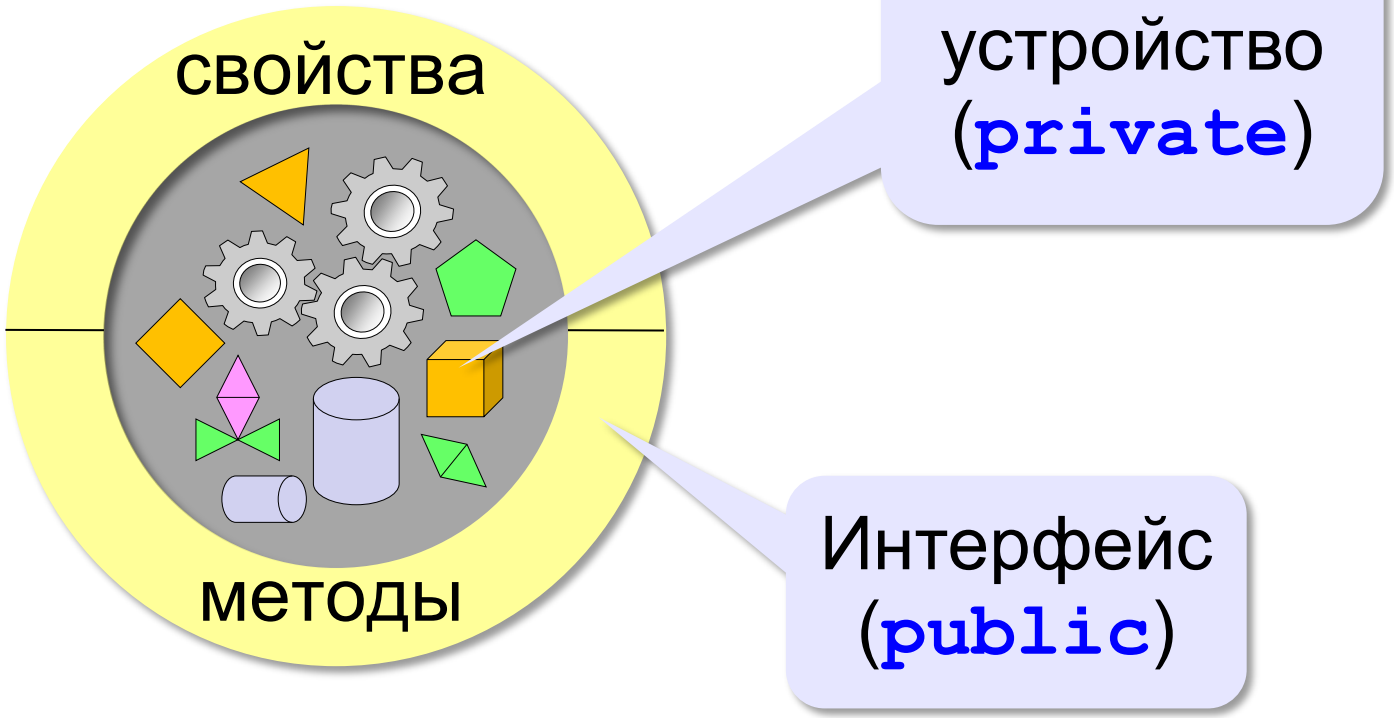
Инкапсуляция

Объектная модель задачи

- Инкапсуляция выполняется посредством сокрытия внутренней информации, то есть маскировкой всех внутренних деталей, не влияющих на внешнее поведение

Инкапсуляция – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение

Инкапсуляция



Инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации

Инкапсуляция

oop04a.py

- По умолчанию атрибуты в классах являются общедоступными
- Инкапсуляция предотвращает прямой доступ к атрибутам объект из вызывающего кода

Атрибут может быть объявлен приватным (внутренним) с помощью двойного нижнего подчеркивания перед именем

```
class B2:
    __count = 0

    def __init__(self):
        B2.__count += 1

    def __del__(self):
        B2.__count -= 1

    def get_count():
        return B2.__count

    def set_count(count):
        B2.__count = count
```

```
print(B2.__count)     # AttributeError: type object 'B2' has no attribute '__count
```

Инкапсуляция – метод __setattr__()

oop04a.py

- Можно запретить назначать атрибуты объекту за пределами класса

```
class A:
    def __init__(self, v):
        self.field1 = v

    def __setattr__(self, attr, value):
        if attr == 'field1':
            self.__dict__[attr] = value
        else:
            raise AttributeError

a = A(1)
a.field2 = 2
```

С помощью метода перегрузки оператора присваивания атрибуту __setattr__():

При попытке создать новое поле возбуждается исключение

Новое поле создать будет нельзя (возбуждается исключение)

Инкапсуляция

- Скрыть атрибуты класса – сделать их приватными и ограничить доступ к ним

С помощью специальных методов

oop04.py

```
class Person:

    # конструктор
    def __init__(self, name):
        self.__name = name # приватный атрибут
        self.__age = 10

    # геттер
    def get_age(self):
        return self.__age

    # сеттер
    def set_age(self, value):
        if value in range(1, 100):
            self.__age = value
        else:
            print("Недопустимый возраст")
```

Инкапсуляция

oop05.py

- Использование аннотаций, которые предваряются символом @
  - Для создания свойства-геттера над свойством ставится аннотация **@property**
  - Для создания свойства-сеттера над свойством устанавливается аннотация **имя_свойства_геттера.setter**

```
class Person:

    # конструктор
    def __init__(self, name):
        self.__name = name  # имя
        self.__age = 1      # возраст

    @property   # Для создания свойства-геттера
    def age(self):
        return self.__age

    @age.setter  # Для создания свойства-сеттера
    def age(self, age):
        if age in range(1, 100):
            self.__age = age
        else:
            print("Недопустимый возраст")

    @property
    def name(self):
        return self.__name
```

```
# создание объектов класса
person3 = Person("Петр")
person3.age = 77   # обращение к сеттер
person3.name = "Вася"  # AttributeError:
```

ЛЕКЦИЯ 4. ОТНОШЕНИЯ МЕЖДУ КЛАССАМИ

Учебные вопросы:

1. Зависимость.
2. Наследование.
3. Композиция и агрегация.
4. Определение (перегрузка) операторов.

Отношения между классами

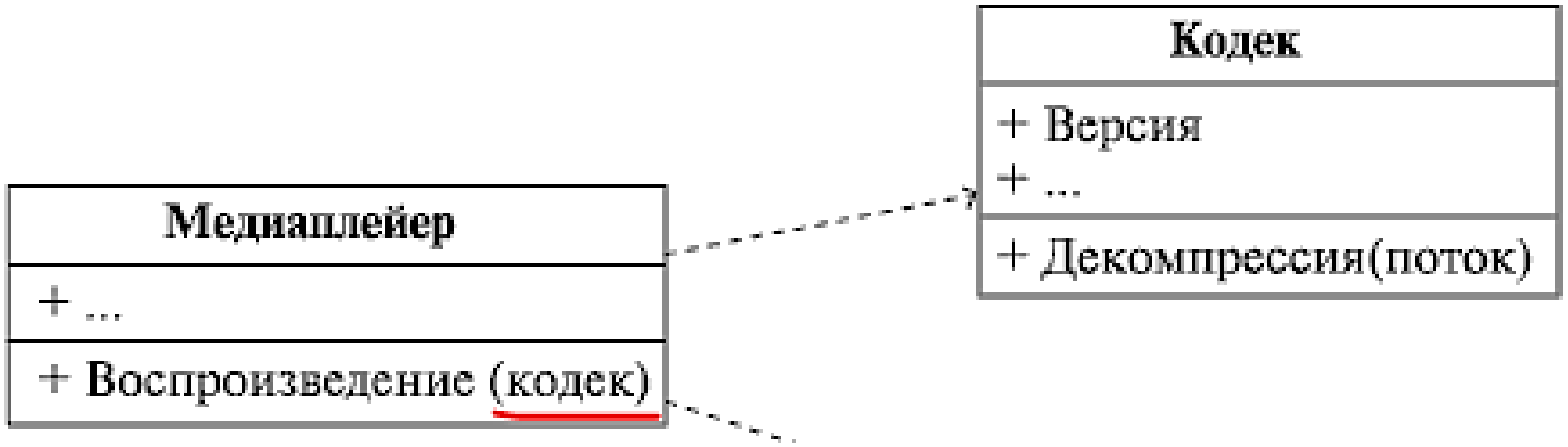
- Зависимость
- Обобщение (наследование)
- Реализация
- Ассоциация
  - Агрегация
  - Композиция

Зависимость

- Зависимость (dependency) — однонаправленное отношение использования между двумя классами:
  - На одном конце отношения находится **зависимый** класс, на втором — **независимый**
- Объект-клиент зависимого класса для своего корректного функционирования *пользуется услугами* объекта-сервера независимого класса
- Зависимость отражает связь между объектами по применению, когда изменение поведения сервера может повлиять на поведение клиента
- Зависимость – не структурная связь

Зависимость. Пример

- Операция "*Воспроизведение*", реализуемая программой-медиаплеером, зависит от операции "*Декомпрессия*", реализуемой кодеком



Зависимость. Пример

- Операция "*Бросок*", реализуемая Игроком, зависит от операции "*Бросок(bro)*", реализуемой игральной костью

GamerKub(class).py

```
class Dice:
    def __init__(self):
        self.n = randint(1, 6)

    def bro(self):
        self.n = randint(1, 6)


class Gamer:

    def __init__(self, name, n = 0):
        self.name = name

    def brosok(self, dice):
        dice.bro()
        self.n = dice.n
        return self.n

    def __str__(self):
        return "Игрок {0.name}".format(self)
```

Создается игрок

Создается игральная кость

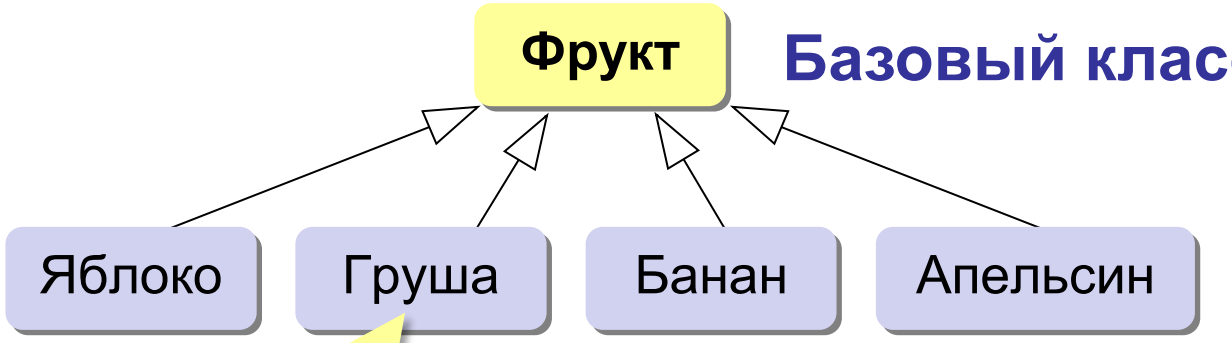
```
g1 = Gamer(igrok1)
g2 = Gamer(igrok2)
d1 = Dice();

n1 = g1.brosok(d1)
print('Выпало:', n1)
```

Игрок бросает кость

Наследование

- Наследование означает такое отношение между классами (отношение родитель/потомок), когда один класс заимствует структурную или функциональную часть другого класса
- Наследование создает такую иерархию абстракций, в которой подклассы наследуют строение от суперкласса:
  - медведь *есть (является)* млекопитающее,
  - дом *есть (является)* недвижимость,
  - "быстрая сортировка" *есть (является)* сортирующий алгоритм.

Наследование



Базовый класс

Классы-наследники

это **фрукт**,
у которого…

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)

Расширение базовых классов

Производный класс (**Базовый класс**)

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

- ✓ Имя *BaseClassName* должно быть определено в пределах области видимости производного класса
- ✓ Допускается, когда базовый класс определен в другом модуле:

```
class DerivedClassName(modname.BaseClassName):
```

Расширение базовых классов

Производный класс

Базовый класс

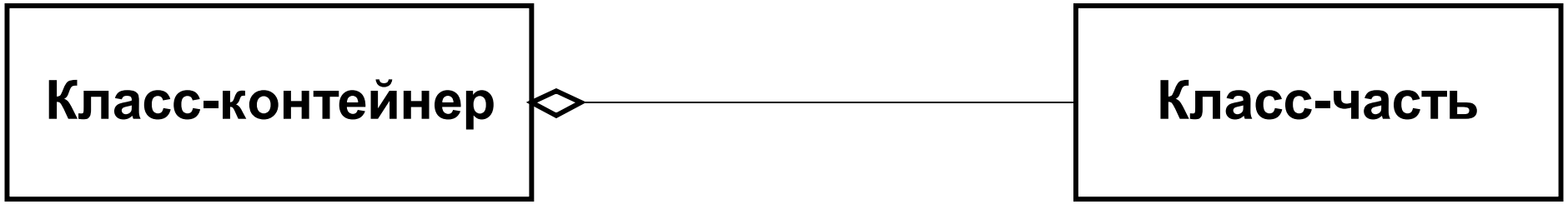
Класс Circle добавляет один атрибут данных (radius) и три новых метода

Класс Circle переопределяет несколько методов класса Point

```
class Circle(Point):

    def __init__(self, radius, x=0, y=0):
        super().__init__(x, y)
        self.radius = radius

    def edge_distance_from_origin(self):
        return abs(self.distance_from_origin() - self.radius)

    def area(self):
        return math.pi * (self.radius ** 2)

    def circumference(self):
        return 2 * math.pi * self.radius

    def __eq__(self, other):
        return self.radius == other.radius and super().__eq__(other)

    def __repr__(self):
        return "Circle({0.radius!r}, {0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return repr(self)
```

Агрегация *(aggregation)*

- Направленное отношение между двумя классами, предназначенное для представления ситуации, когда один из классов представляет собой некоторую сущность, которая включает в себя в качестве составных частей другие сущности



- Агрегация является частным случаем ассоциации

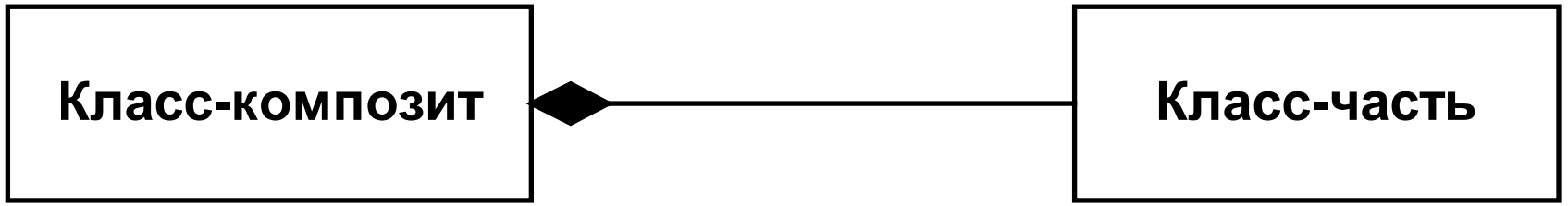
Агрегация *(aggregation)*

```
class Salary():
  def __init__(self, pay):
    self.pay = pay

  def getTotal(self):
    return (self.pay * 12)

class Employee():
  def __init__(self, pay, bonus):
    self.pay = pay
    self.bonus = bonus

  def annualSalary(self):
    return "Total: " + str(self.pay.getTotal() + self.bonus)



if __name__ == "__main__":
  salary = Salary(100)
  employee = Employee(salary, 10)
  print(employee.annualSalary())
```

При создании объекта *employee* ему передается новый *salary*

Композиция *(composition)*

- *Композиция* предназначена для спецификации более сильной формы отношения "часть-целое", при которой с уничтожением объекта класса-контейнера уничтожаются и все объекты, являющимися его составными частями.



- Композиция является частным случаем ассоциации
- Разновидность физического включения означает, что объект класса-часть не существует отдельно от объемлющего экземпляра класса-композита

Композиция *(composition)*

```
class Salary:
  def __init__(self,pay):
    self.pay = pay

  def getTotal(self):
    return (self.pay*12)

class Employee:
  def __init__(self,pay,bonus):
    self.pay = pay
    self.bonus = bonus
    self.salary = Salary(self.pay)

  def annualSalary(self):
    return "Total: " + str(self.salary.getTotal() + self.bonus)


if __name__ == "__main__":
  employee = Employee(100,10)
  print(employee.annualSalary())
```

При создании объекта *employee* ему создается свой *salary*

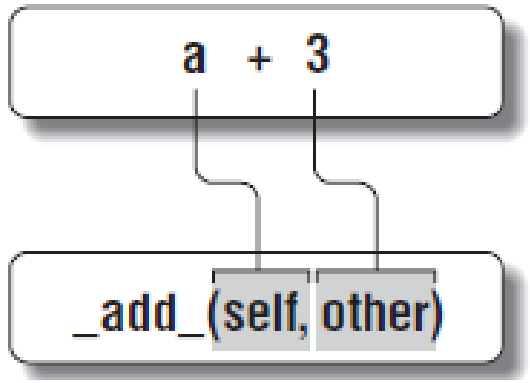
Специальные методы

- Методы, имена которых обрамляются __, Python трактует как *специальные*, например, __init__ (инициализация) или __str__ (строковое представление).
- Специальные методы, как правило, идут первыми при объявлении класса.

```
class Point:
      def __init__ (self, x=0, y=0):
            self.x = x
            self.y = y
      def __eq__(self, other):
            return self.x == other.x and self.y == other.y
      def __repr__(self):
            return "Point({0.x!r}, {0.y!r})".format(self)
      def __str__(self):
            return "({0.x! r}, {0.y!r})".format(self)
      def distance_from_origin(self):
            return math.hypot(self.x, self.y)
```

Определение (перегрузка) операторов

- Перегрузка операторов
  - позволяет объектам, созданным из классов, перехватывать и участвовать в операциях, которые применяются к встроенным типам
  - реализуется за счет создания методов со специальными именами для перехватывания операций
- Пример.
  - Для реализации операции сложения (+) объект экземпляра должен наследовать метод `__add__`
  - Этот метод будет вызываться всякий раз, когда объект будет появляться в выражении с оператором “+”



Возвращаемое значение метода становится результатом операции

```
class Point2D:
    . . .
    def __add__(self, other):
        return Point2D(self.x + other.x, self.y + other.y)
```

Определение (перегрузка) операторов

Перед действием рекомендуется проверить, экземпляром какого класса является переданный объект

```
class Point2D:

    def __add__(self, other):
        """Создать новый объект как сумму координат self и other.
        с проверкой типа передаваемого объекта"""

        if isinstance(other, self.__class__):
            # Точка с точкой
            # Возвращаем новый объект!
            return Point2D(self.x + other.x, self.y + other.y)
        elif isinstance(other, (int, float)):
            # Точка и число
            # Добавим к обеим координатам self число other и вернем результат
            # Возвращаем старый, измененный, объект!
            self.x += other
            self.y += other
            return self
        else:
            # В противном случае возбуждаем исключение
            raise TypeError("Не могу добавить {1} к {0}".
                            format(self.__class__, type(other)))
```

Пример.

```
Inheritor...
in Super.method

Provider

Replacer...
in Replacer.method

Provider

Extender...
starting Extender.method
in Super.method
ending Extender.method

Provider
in Provider.action1
```

Super – абстрактный суперкласс?

oop09.py

```
class Super:
    def method(self):
        print('in Super.method')       # Стандартное поведение
    def delegate(self):
        self.action()                  # Ожидается определение метода

class Inheritor(Super):
    pass                               # Буквальное наследование метода

class Replacer(Super):
    def method(self):
        print('in Replacer.method')    # Полное замещение метода

class Extender(Super):
    def method(self):                  # Расширение поведения метода
        print('starting Extender.method')
        Super.method(self)
        print('ending Extender.method')

class Provider(Super):                 # Заполнение обязательного метода
    def action(self):
        print('in Provider.action1')

if __name__ == '__main__' :
    for klass in (Inheritor, Replacer, Extender):
        print ('\n' + klass.__name__      + '...')
        klass().method()
        print('\nProvider' )
    x = Provider()
    x.delegate()
```

Пример.

oop09.py

```
class Super:
    def method(self):
        print('in Super.method')        # Стандартное поведение
    def delegate(self):
        self.action()                   # Ожидается определение метода

class Inheritor(Super):
    pass                                # Буквальное наследование метода

class Replacer(Super):
    def method(self):
        print('in Replacer.method')     # Полное замещение метода

class Extender(Super):
    def method(self):                   # Расширение поведения метода
        print('starting Extender.method')
        Super.method(self)
        print('ending Extender.method')

class Provider(Super):                  # Заполнение обязательного метода
    def action(self):
        print('in Provider.action1')

if __name__ == '__main__' :
    for klass in (Inheritor, Replacer, Extender):
```

```
lek_3-4\oop09.py", line 5, in delegate
    self.action()                   #     Ожидается определение метода
AttributeError: 'Extender' object has no attribute 'action'
```

```
    x = Extender()
    x.delegate()
```

Если ожидаемый метод в подклассе не определен, тогда после неудавшегося поиска при наследовании Python *генерирует исключение*