

Основные парадигмы программирования

Введение в объектно-
ориентированное
программирование

К чему приведет процедурный подход?

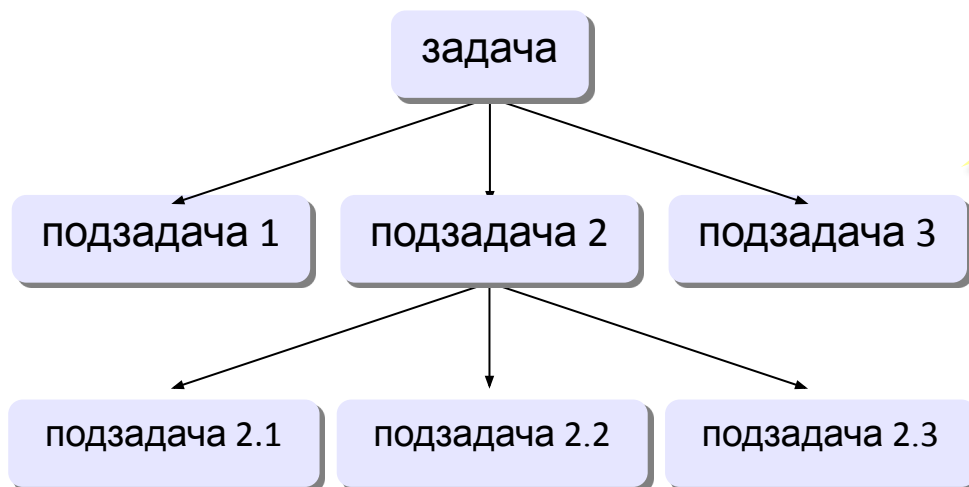


Главная проблема – **сложность!**

- программы из миллионов строк
- тысячи переменных и массивов

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

Структурное программирование:



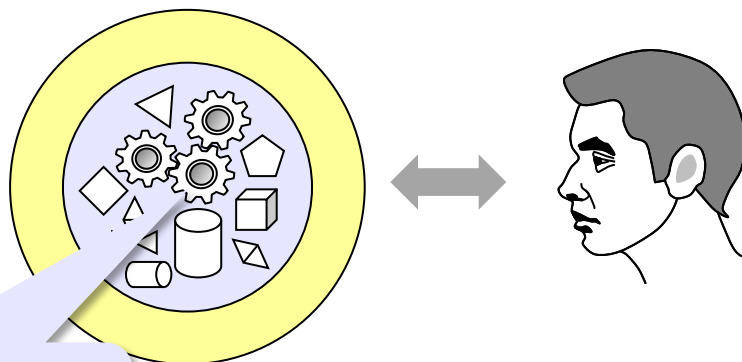
декомпозиция по задачам



человек мыслит иначе, объектами

Восприятие внешнего мира

Как мы воспринимаем объекты?



существенные
свойства

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



Разные цели – **разные модели!**

Основная идея

- Разработка объектно-ориентированного ПО исходит из осознания того, что правильно построенные системы программной инженерии должны основываться на повторно используемых **компонентах** высокого качества, как это делается в других инженерных сферах

ОО-подход определяет, какую форму должны иметь эти компоненты: каждый из них должен быть основан на некотором типе объектов



Объектный подход

- OOA (object oriented analysis)
объектно-ориентированный анализ
- OOD (object oriented design)
объектно-ориентированное проектирование
- OOP (object oriented programming)
объектно-ориентированное
программирование

Объектно-ориентированный анализ

- Выделить объекты
- Определить их существенные свойства
- Описать поведение (команды, которые они могут выполнять)

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

- Состояние определяет поведение
 - лежащий человек не прыгнет
 - незаряженное ружье не выстрелит

Объектная модель

Для объектно-ориентированного стиля
концептуальная база - это **объектная модель**

- объектом может быть:

- автомобиль,
- человек и т.д.

Что делает объект объектом?

- объекты обладают:

- цвет,
- размер и т.д.

Не то, что он может иметь
физического двойника,
а то, что с ним можно

- они обладают поведением:

- начинают функционировать
- меняют свое состояние
- имеют набор внешних входов и выходов

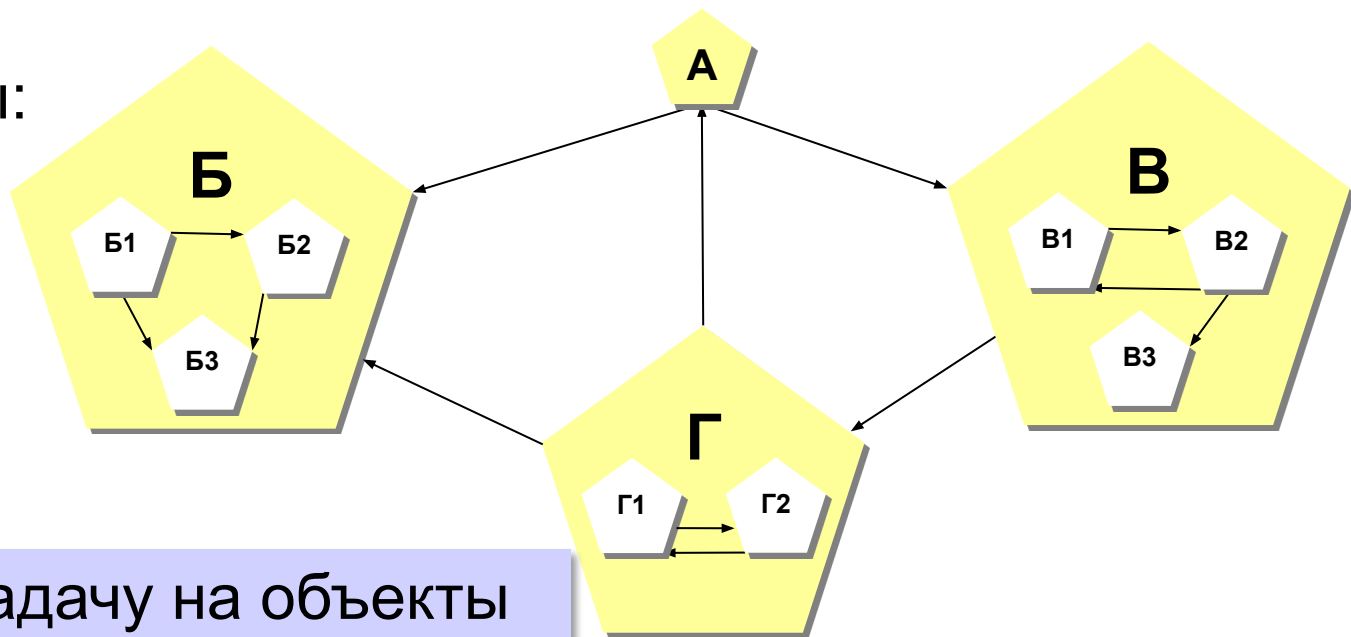
манипулировать в программе,
используя множество хорошо
определенных операций,
называемых методами

Объектно-ориентированный подход

- Основополагающая идея:
 - объединение **данных** и **действий**, производимых над **данными**, в единое целое, которое называется **объектом**

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

- Структура программы:



«Разделить» задачу на объекты

Объектно-ориентированная декомпозиция

- Предметная область представлена как совокупность некоторых автономных объектов, которые взаимодействуют друг с другом, чтобы обеспечить функционирование всей системы в целом



Объектно-ориентированный анализ и проектирование (ООАП)

- В процессе ОО-анализа основное внимание уделяется определению и описанию объектов (понятий) в терминах предметной области
 - Пример, в библиотечной информационной системе среди понятий должны присутствовать *Book* (книга), *Library* (библиотека) и *Patron* (клиент)
- В процессе ОО-проектирования определяются программные объекты и способы их взаимодействия с целью выполнения системных требований
 - Пример, в библиотечной системе программный объект *Book* может содержать атрибут *title* (название) и метод *getChapter* (определить номер главы)

Объектно-ориентированный анализ и проектирование (ООАП)

- Пример. Объект «**Маршрут**»
- Вопросы, которые можно задавать:
 - Какова начальная точкой маршрута? Какова конечная точка?
 - Как передвигаемся на маршруте: пешком, на автобусе, на автомобиле, метро или маршрут смешанный?
 - Сколько этапов включает маршрут?
 - Какие линии метро используются, если они есть в маршруте?
- Запросы (queries) – методы, позволяющие получать свойства объекта
- Команды (commands) – позволяют изменять видимые (доступные) свойства объектов
 - Удалить (этап), Присоединить (этап), Добавить в начало (этап)

Объектно-ориентированный анализ и проектирование (ООАП)

- Требуется добавить **методы программных классов**, описывающие передачу сообщений между объектами для удовлетворения требованиям
 - ✓ Вопрос определения способов взаимодействия объектов и принадлежности методов важен и не тривиален
- Применить принципы и шаблоны объектного проектирования для создания проектных моделей взаимодействия объектов
 - шаблоны проектирования **GRASP** – шаблоны распределения обязанностей

Проектирование на основе обязанности

- Программные объекты имеют обязанности
- В UML обязанность (responsibility) определяется как

“контракт или обязательство”

- Под обязанностью в контексте GRASP понимается некое *действие (функция)* объекта
- Обязанности описывают поведение объекта
- В общем случае два типа обязанностей:
 - Знание (knowing)
 - Действие (doing)

Проектирование на основе обязанностей
responsibility-driven design — RDD

Обязанности, относящиеся к действиям объекта

- Выполнение некоторых действий самим объектом, например, создание экземпляра или выполнение вычислений.
- Инициирование действий других объектов.
- Управление действиями других объектов и их координирование

Пример.

- Объект Sale отвечает за создание экземпляра SalesLineItems (действие)

Обязанности, относящиеся к знаниям объекта

- Наличие информации о закрытых инкапсулированных данных.
- Наличие информации о связанных объектах.
- Наличие информации о следствиях или вычисляемых величинах.

Пример.

- Объект Sale отвечает за наличие информации о стоимости покупки (знание)

Реализация обязанностей

- Обязанности реализуются посредством **методов**, действующих либо отдельно, либо во взаимодействии с другими методами и объектами
- Пример.
 - Для класса `Sale` можно определить один или несколько методов вычисления стоимости (метод `getTotal`).
 - Для выполнения этой обязанности объект `Sale` должен взаимодействовать с другими объектами, в том числе передавать сообщения *getSubtotal* каждому объекту `SalesLineItem` о необходимости предоставления соответствующей информации этими объектами

Принципы и рекомендации ООАП

- GRASP – General Responsibility Assignment Software Patterns (Общие шаблоны распределения обязанностей в программных системах)
 - Information Expert
 - информационный эксперт, класс, у которого имеется информация, требуемая для выполнения обязанности
 - Creator
 - Назначить классу В обязанность создавать экземпляры класса А

Идеальный класс должен иметь лишь одну причину для изменения, обладать минимальным интерфейсом, правильно реализовывать наследование и предотвращать каскадные изменения в коде при изменении требований

Принципы и рекомендации ООАП

- GRASP – General Responsibility Assignment Software Patterns (Общие шаблоны распределения обязанностей в программных системах)
 - High Cohesion
 - Распределение обязанностей, поддерживающее высокую степень зацепления
 - Low Coupling
 - Распределить обязанности таким образом, чтобы степень связанности оставалась низкой
 - Controller
 - Делегирование обязанностей по обработке системных сообщений другому классу

Объектно-ориентированное программирование

- Объектно-ориентированное программирование – это методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является **экземпляром** определенного **класса**, а классы образуют иерархию наследования
- Особенности:
 - ООП использует в качестве базовых элементов объекты, а не алгоритмы
 - каждый объект является экземпляром какого-либо определенного класса (играющего роль типа данных)
 - классы организованы иерархически

При ООП подходе: раскладываем программный код на составляющие, чтобы уменьшить его избыточность, и пишем новый код, адаптируя имеющийся программный код, а не изменяя его.

Принципы и рекомендации ООАП

■ Принципы SOLID

- Single Responsibility Principle (Принцип единственной обязанности)
- Open/Closed Principle (Принцип открытости/закрытости)
- Liskov Substitution Principle (Принцип подстановки Лисков)
- Interface Segregation Principle (Принцип разделения интерфейсов)
- Dependency Inversion Principle (Принцип инверсии зависимостей)

Принципы и рекомендации ООАП

■ Шаблоны GoF (Gang-of-Fou)

□ Структурные паттерны

- Применяются при компоновке системы на основе классов и объектов (Adapter, Facade, Decorator, Proxy)

□ Порождающие паттерны

- Предназначены для создания объектов, позволяя системе оставаться независимой как от самого процесса порождения, так и от типов порождаемых объектов (Factory Method, Abstract Factory,

□ Паттерны поведения

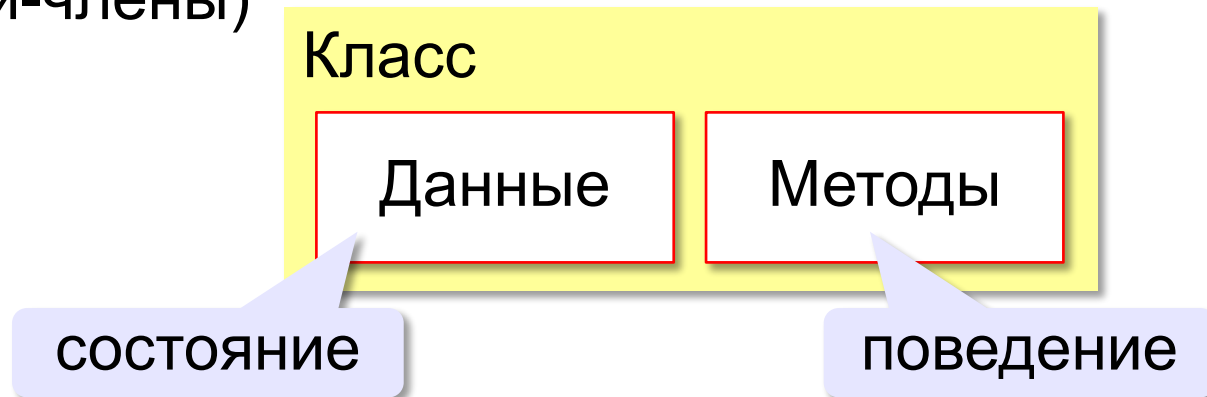
- Описывают правильные способы организации взаимодействия между используемыми объектами (Template Method, Strategy)

Классы и объекты

- Класс определяет
 - данные (переменные)
 - поведение (методы).
 - ✓ данные и методы класса также называют членами класса.
- Класс рассматривается как определяемый пользователем тип данных.
- Объектом называется экземпляр некоторого класса.
 - Объект создается как переменная типа класса, которая используется для доступа к данным - членам класса и для вызова методов - членов класса.

Класс

- Класс — это структура данных, объединяющая состояние (поля) и действия (методы и другие функции-члены)



- Класс предоставляет определения для динамически создаваемых **экземпляров** класса (**объектов класса**)

- Классы поддерживают механизмы наследования и полиморфизма, которые позволяют создавать производные классы, расширяющие функциональные возможности базового класса

Класс

- Класс — это структура данных, объединяющая состояние (поля) и действия (методы и другие функции-члены).

```
class Complex
{
    private:
        int real;        // вещественная часть
        int imaginary;    // мнимая часть

    public:
        Complex Add(Complex x);

};
```



Определение класса

- Класс определяется с помощью ключевого слова **class**
- Создается новое пространство имен и используется в качестве локальной области видимости при выполнении инструкций в теле определения

```
class ИмяКласса:  
    код_тела_класса
```

По окончании выполнения определения создается объект
(объект-класс)

Определение класса

Это объект и поэтому:

- его можно присвоить переменной,
- его можно скопировать,
- можно добавить к нему атрибут,
- его можно передать функции в качестве аргумента

Можно поместить определение класса в одну из ветвей инструкции *if* или в тело функции

```
if paramF > 0:  
    class Person:  
        lev = "1 уровень" # атрибут  
        класса  
        def display_info():  
            print("Level: ", Person.lev)  
else:  
    ...
```

Классы и объекты

- Создание экземпляра класса использует запись вызова функций
- Созданный объект связывают с переменной:
имя_объекта = ИмяКласса([параметры])

```
person1 = Person()  
person1.display_info()
```

Если у объекта **person1** нет своего собственного метода **display_info()**, он ищется в классе **Person**

Пример. Класс. Точка на плоскости

```
class Point:  
    def __init__(self, x=0, y=0):  
        self.x = x  
        self.y = y
```

Конструктор

```
    def distance_from_origin(self):  
        return math.hypot(self.x, self.y)
```

Свой метод

```
    def __eq__(self, other):  
        return self.x == other.x and self.y == other.y
```

```
    def __str__(self):  
        return "({0.x!r}, {0.y!r})".format(self)
```

Стандартные методы

Класс Point (сохранен в файле Shape.py)

```
import Shape
a = Shape.Point()
b = Shape.Point(3, 4)
str(b)           # вернет: '(3, 4)'
b.distance_from_origin() # вернет: 5.0
b.x = -19
str(b)           # вернет: '(-19, 4)'
a == b, a != b   # вернет: (False, True)
```

- Когда создается объект (`p = Shape. Point()`), то сначала вызывается специальный метод `new()` (этот метод реализован в базовом классе `object`), который создает объект, а затем выполняется инициализация объекта вызовом специального метода `init()`.

Пример. Стек на базе списка

```
class Stack:
    def __init__(self):
        """Инициализация стека"""
        self._stack = []

    def top(self):
        """Возвратить вершину стека (не удаляя элемент)"""
        return self._stack[-1] # -1 индекс последнего элемента

    def pop(self):
        """Снять со стека элемент"""
        return self._stack.pop()

    def push(self, x):
        """Поместить элемент на стек"""
        self._stack.append(x)

    def __len__(self):
        """Количество элементов в стеке"""
        return len(self._stack)

    def __str__(self):
        """Представление в виде строки"""
        return " ; ".join(["%s" % e for e in self._stack])
```

Список

Метод списка

Метод списка

Атрибуты и методы

Print output (drag lower right corner to resize)

```
pat.course 6.01
pat.building 34
pat.building 32
```

```
class Staff601:
    course = '6.01'
    building = 34
    room = 501

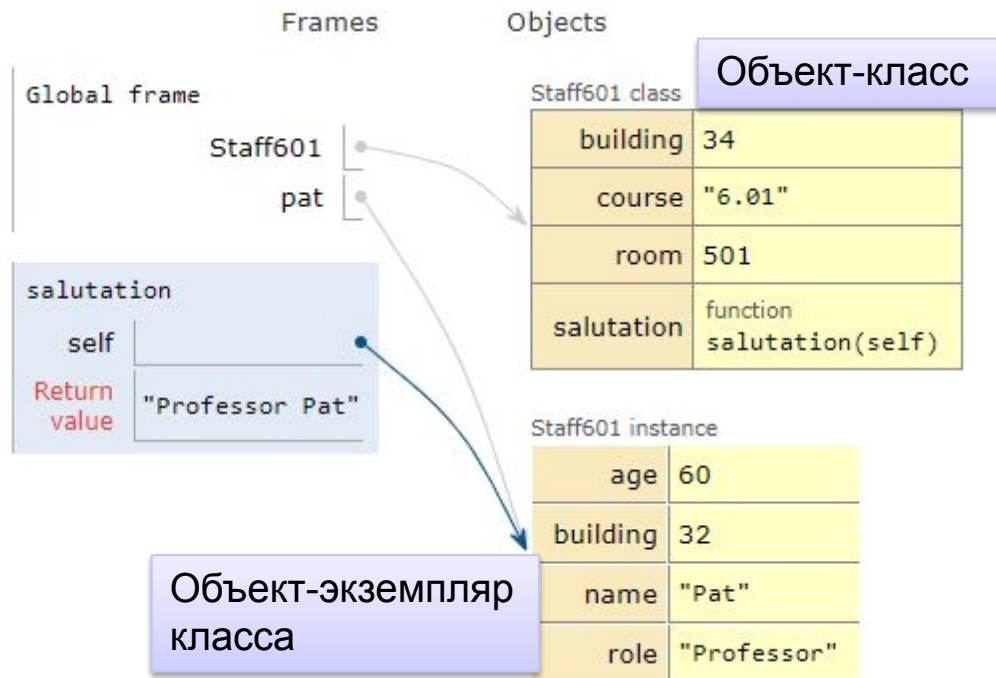
    def salutation(self):
        return self.role + ' ' + self.name
```

```
pat = Staff601()
print("pat.course", pat.course)
```

```
pat.name = 'Pat'
pat.age = 60
pat.role = 'Professor'
```

```
print("pat.building", pat.building)
pat.building = 32
print("pat.building", pat.building)
```

```
print("pat.salutation()", pat.salutation())
print("Staff601.salutation(pat)", Staff601.salutation(pat))
```



Атрибуты и методы

Print output (drag lower right corner to resize)

```
pat.course 6.01
pat.building 34
pat.building 32
pat.salutation() Professor Pat
Staff601.salutation(pat) Professor Pat
```

```
class Staff601:
    course = '6.01'
    building = 34
    room = 501

    def salutation(self):
        return self.role + ' ' + self.name
```

```
pat = Staff601()
print("pat.course", pat.course)
```

```
pat.name = 'Pat'
pat.age = 60
pat.role = 'Professor'
```

```
print("pat.building", pat.building)
pat.building = 32
print("pat.building", pat.building)
```

```
print("pat.salutation()", pat.salutation())
print("Staff601.salutation(pat)", Staff601.salutation(pat))
```

Метод
объекта-
экземпляра

Frames

Global frame	
Staff601	•
pat	•

Objects

Staff601 class

building	34
course	"6.01"
room	501
salutation	function salutation(self)

Staff601 instance

age	60
building	32
name	"Pat"
role	"Professor"

Атрибуты

oop01.py

- Атрибуты хранятся в специальном словаре и к нему можно обратиться по имени `__dict__`:

`Nclass.__dict__`

`ob.__dict__`

- Если обратиться к атрибуту, которого не существует, то будет возбуждено исключение *AttributeError*

Атрибуты

oop01.py

- `hasattr(obj, attr_name)` - проверить наличие атрибута `attr_name` в объекте `obj`.
 - Если атрибут присутствует, то функция возвращает `True`, иначе `False`.
- `getattr(obj, attr_name[, default_value])` - получить значение атрибута `attr_name` в объекте `obj`.
 - Если атрибут не был найден, то будет возбуждено исключение `AttributeError`. Можно указать значение по умолчанию `default_value`, которое будет возвращено, если атрибута не существует.
- `setattr(obj, attr_name, value)` - изменить значение атрибута `attr_name` на `value`. Если атрибут не существовал, то он будет создан.

Конструктор

- Конструктором класса называют метод, который автоматически вызывается при создании объектов
- По умолчанию создается автоматически

```
class Person:  
    name = "Иван"  
  
p = Person()  
print(p.name) # Иван
```

ВЫЗОВ
конструктора

Конструктор

- В Python роль конструктора играет метод `__init__()`
- Необходимость конструкторов связана с тем, что часто объекты должны иметь собственные свойства сразу
- Конструктор класса не позволит создать объект без обязательных полей
- Первый его параметр – **self** – ссылка на сам только что созданный объект, остальные параметры – атрибуты объекта

ВЫЗОВ
конструктора

```
class Person:  
    def __init__(self, name):  
        self.name = name  
  
person1 = Person("Иван")
```

Конструктор

- В Python роль конструктора играет метод `__init__()`
- Если надо допустить создание объекта, даже если никакие данные в конструктор не передаются, то в таком случае параметрам конструктора класса задаются *значения по умолчанию*
- Первый его параметр – **self** – ссылка на сам только что созданный объект, остальные параметры – атрибуты объекта

ВЫЗОВ
конструктора

```
class Person:
    def __init__(self, name="Иван"):
        self.name = name

person1 = Person("Петр")
person2 = Person()
```

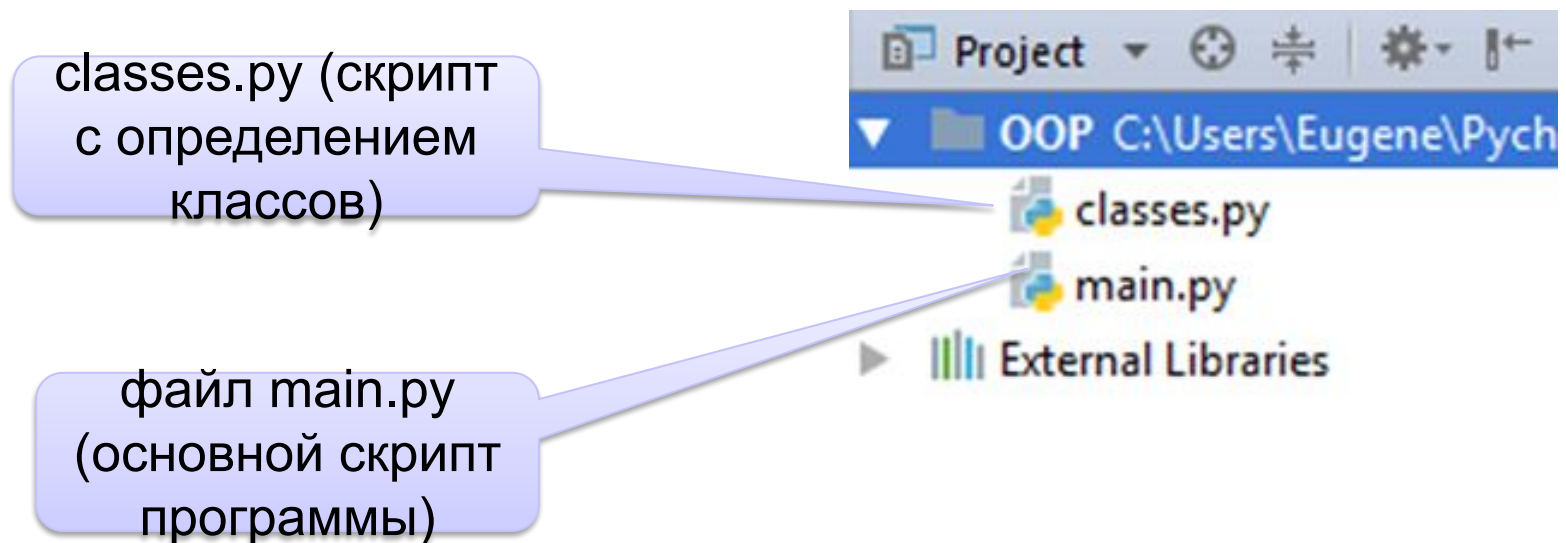
Деструктор

- Деструктор вызывается, когда объект уничтожается
- В языке программирования Python объект уничтожается, когда:
 - исчезают все связанные с ним переменные,
 - им присваивается другое значение, в результате чего связь со старым объектом теряется,
 - он удаляется с помощью команды **del**

```
class Person:  
    # конструктор  
    def __init__(self, name):  
        self.name = name  
    # деструктор  
    def __del__(self):  
        print(self.name, "удален из  
памяти")
```

Определение классов в отдельных модулях

- Как правило, классы размещаются в отдельных модулях и затем импортируются в основной скрипт программы



Перегрузка функций посредством сигнатур вызова?

- Поскольку в Python отсутствуют объявления типов, эта концепция в действительности неприменима – полиморфизм в языке Python основан на интерфейсах объектов, а не на типах

Инструкция `def` просто присваивает объект некоторому имени в области видимости класса и поэтому будет сохранено **только последнее** определение метода

```
class C:  
    def meth(self, x):  
        ...  
    def meth(self, x, y, z):  
        ...
```




Главные элементы объектной модели

- Инкапсуляция
- Иерархия
- Полиморфизм

Инкапсуляция

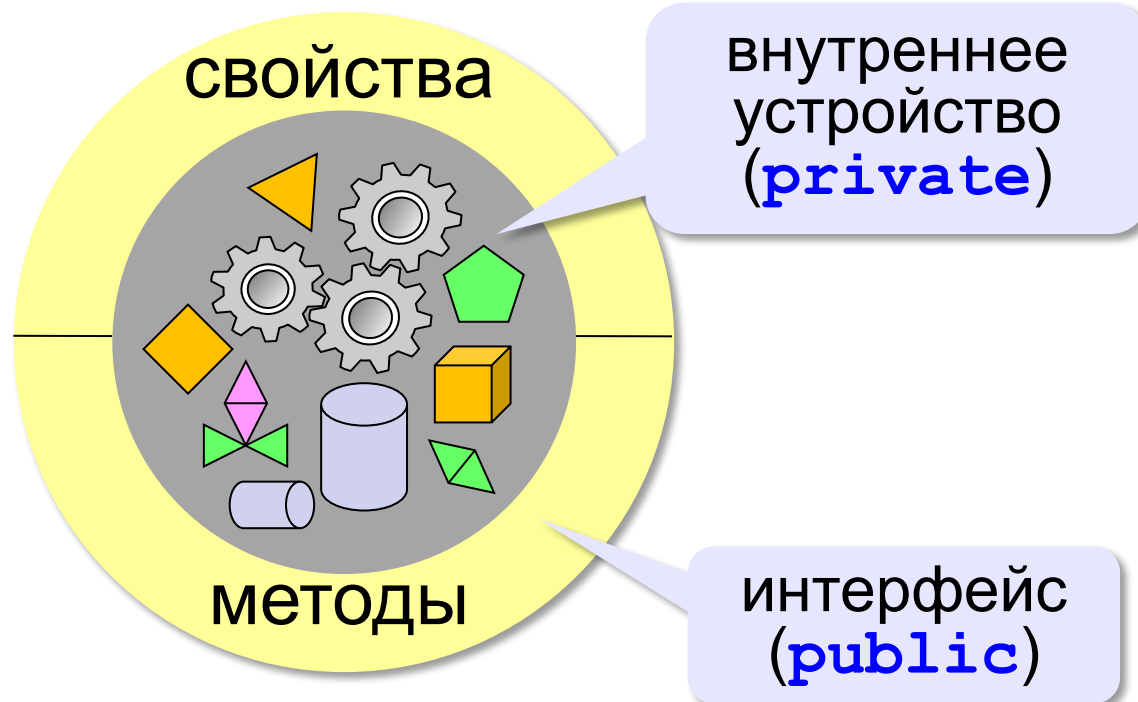
Объектная модель задачи:



- 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

- По умолчанию атрибуты в классах являются общедоступными
- Инкапсуляция предотвращает прямой доступ к атрибутам объекта 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
```

oop04a.py

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

Инкапсуляция – метод `__setattr__()`

oop04a.py

- Можно запретить назначать атрибуты объекту за пределами класса

С помощью метода перегрузки оператора присваивания атрибуту `__setattr__()`:

При попытке создать новое поле возбуждается исключение

```
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
```

Новое поле создать будет нельзя (возбуждается исключение)

Инкапсуляция

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

С помощью специальных методов

```
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")
```

oop04.py

Инкапсуляция

- 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:
```

oop05.py

```
# 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 сеттеру
person3.name = "Вася" # AttributeError
```



Отношения между классами

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

Многоуровневая иерархия

- Точка на плоскости
- DemoPoint
 - Точка в пространстве
 - DemoShape
- Отрезок на плоскости
- DemoLine
 - Треугольник на плоскости
 - DemoTriangle

Вопрос: какие
отношения
должны быть
реализованы между
классами?

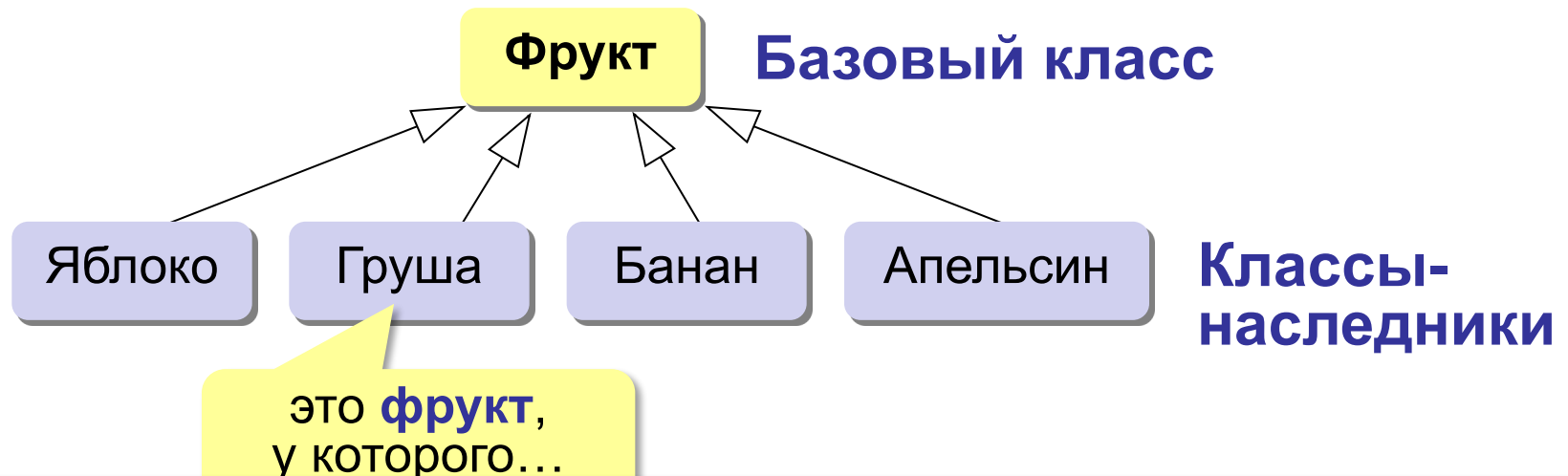
Иерархия

- Иерархия – это упорядочение абстракций – расположение их по уровням.
- Основными видами иерархических структур применительно к сложным системам являются
 - структура классов (иерархия "is-a")
 - структура объектов (иерархия "part of").
- Пример иерархии: **наследование** - основной вид иерархии **"is-a"**

Наследование

- Наследование означает такое отношение между классами (отношение родитель/потомок), когда один класс заимствует структурную и функциональную часть одного или нескольких других классов (соответственно, одиночное и множественное наследование).
- Наследование создает такую иерархию абстракций, в которой подклассы наследуют строение от одного или нескольких суперклассов:
 - медведь есть млекопитающее,
 - дом есть недвижимость,
 - "быстрая сортировка" есть сортирующий алгоритм.

Наследование



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

- ✓ яблоко – фрукт
- ✓ горный клевер – клевер
- ✗ машина – двигатель

яблоко – **это** фрукт

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

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

Расширение базовых классов

```
class Token
```

```
{
```

```
...
```

```
}
```

```
class CommentToken: Token
```

```
{
```

```
...
```

```
}
```

Производный класс

Базовый класс

Двоеточие

Token
« concrete »

CommentToken
« concrete »

Производный класс

Базовый
класс

Расширение базовых классов

```
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)
```

Роль наследования

- Общая часть структуры и поведения сосредоточена в наиболее общем суперклассе.
- Суперклассы отражают наиболее общие, а подклассы - более специализированные абстракции, в которых члены суперкласса могут быть дополнены, модифицированы и даже скрыты.
- Принцип наследования позволяет
 - упростить выражение абстракций,
 - делает проект менее громоздким,
 - более выразительным.

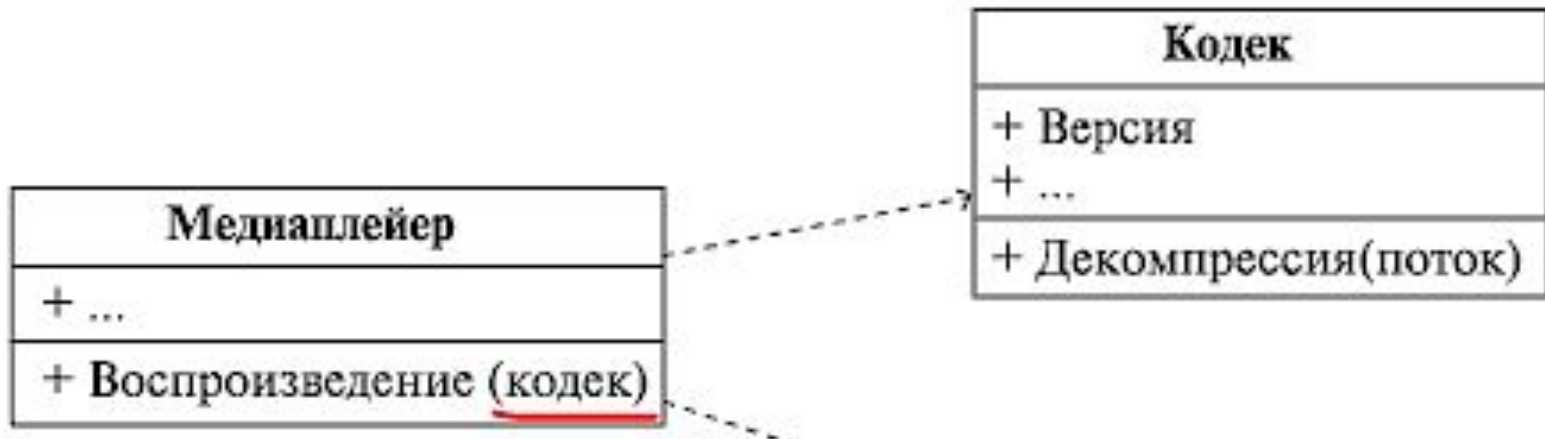


Зависимость

- 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 игровой костью

Создается игрок

Создается игральная кость

Игрок бросает кость

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

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

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)
```

Модель включения/делегации

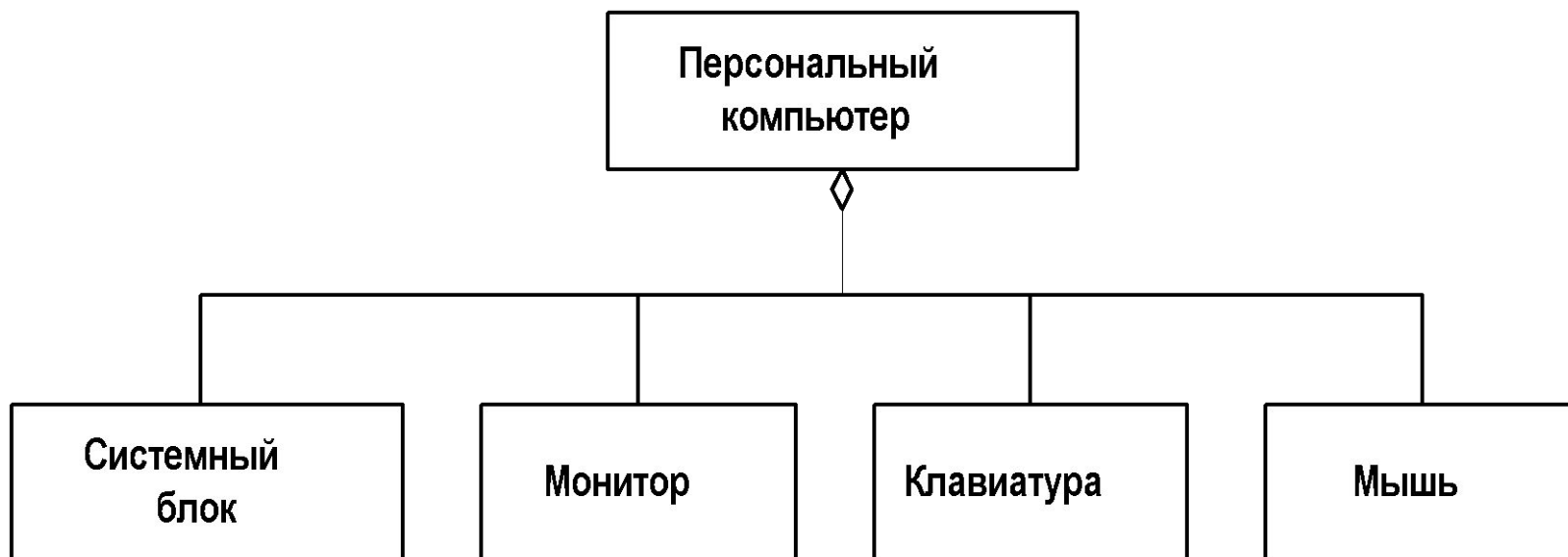
- Реализует отношение “*имеет*” (“*has-a*”) или агрегация.
- Эта форма повторного использования не применяется для установки отношений “родительский-дочерний”.
- Это отношение позволяет одному классу определять переменную-член другого класса и опосредованно представлять его функциональность (при необходимости) пользователю объекта.

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



Пример отношения агрегации



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

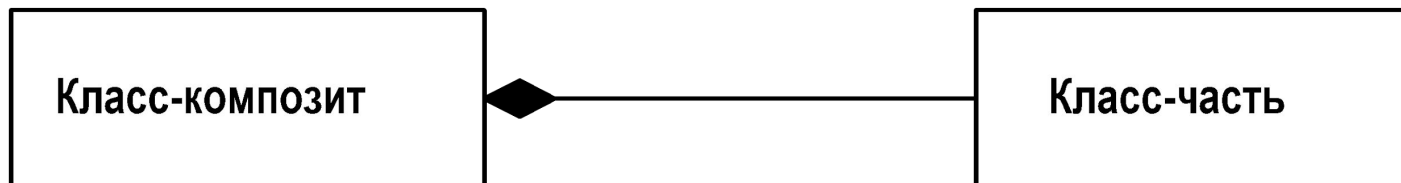
    def annualSalary(self):
        return "Total: " + str(self.pay.getTotal() + self.bonus)

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

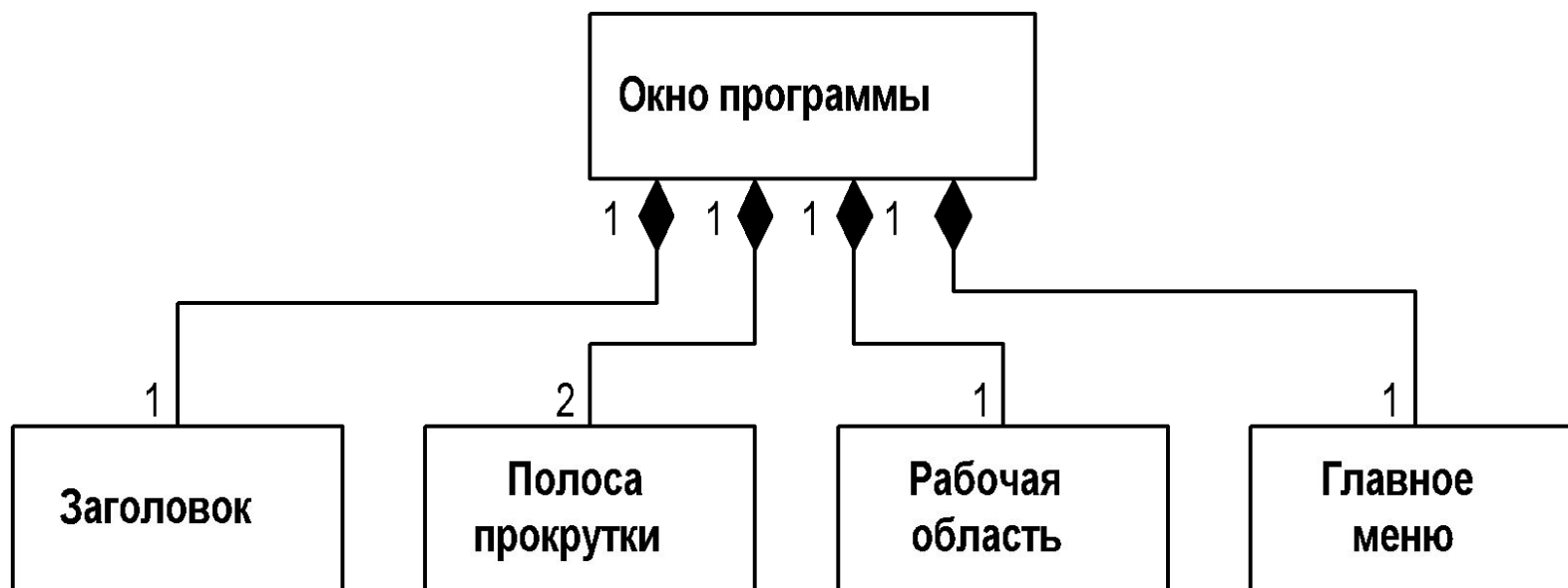
При создании
объекта *employee* ему
передается новый
salary

Композиция (*composition*)

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



Пример отношения композиции



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

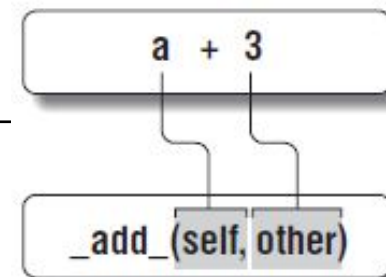
- позволяет объектам, созданным из классов, перехватывать и участвовать в операциях, которые применяются к встроенным типам
- реализуется за счет создания методов со специальными именами для перехватывания операций

■ 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

```
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)))
```

Перед
действием
рекомендуется
проверить,
экземпляром
какого класса
является
переданный
объект

Что хорошего и плохого в ООП?

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