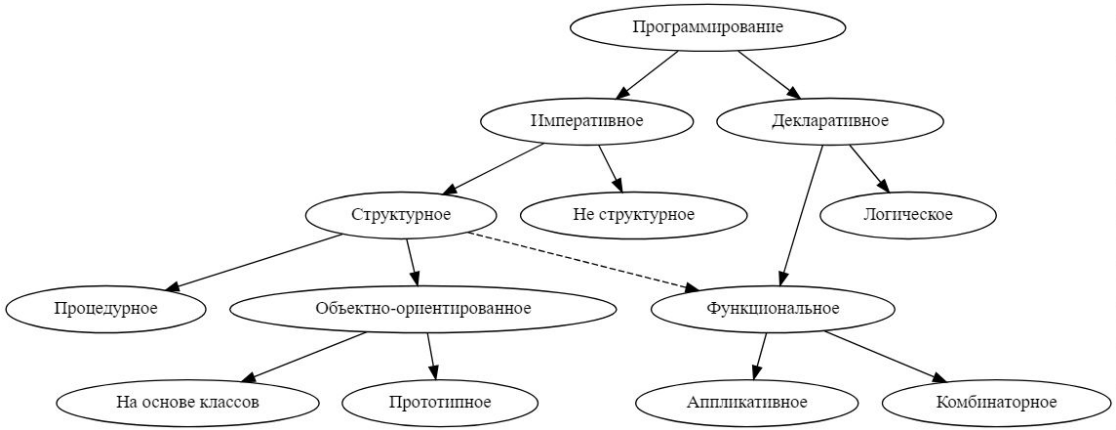
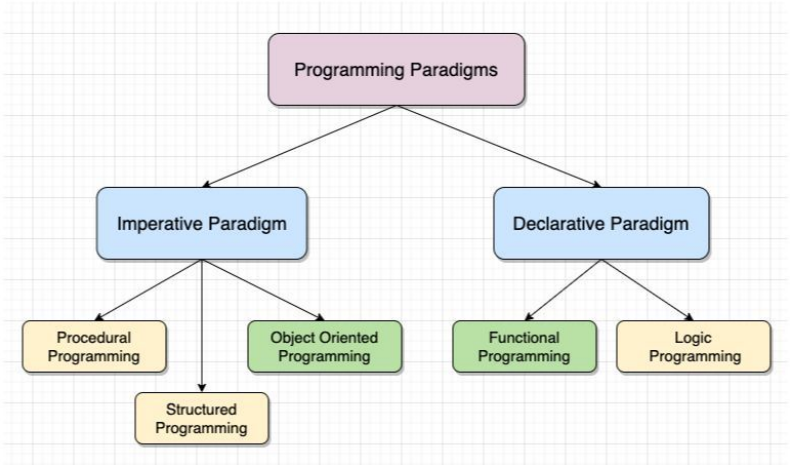
Лекция 7
Объектно-ориентированное программирование.

Парадигмы программирования





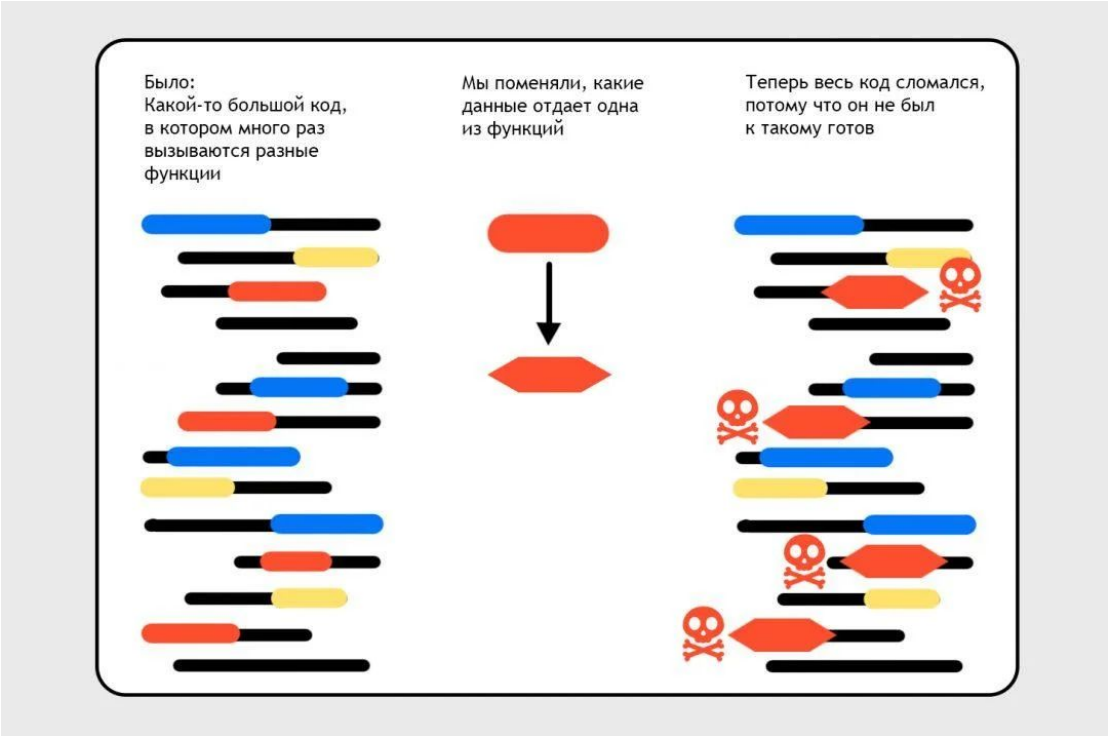
Императивное программирование устроено так:

В языке есть команды, которые этот язык может выполнять. Эти команды можно собрать в подпрограммы, чтобы автоматизировать некоторые однотипные вычисления. В каком порядке записаны команды внутри подпрограммы, в том же порядке они и будут выполняться.
Есть переменные, которые могут хранить данные и изменяться во время работы программы. Переменная — это ячейка для данных. Мы можем создать переменную нужного нам типа, положить туда какое-то значение, а потом поменять его на другое.
Если подпрограмме на вход подать какое-то значение, то результат будет зависеть не только от исходных данных, но и от других переменных. Например, у нас есть функция, которая возвращает размер скидки при покупке в онлайн-магазине. Мы добавляем в корзину товар стоимостью 1000 ₽, а функция должна нам вернуть размер получившейся скидки. Но если скидка зависит от дня недели, то функция сначала проверит, какой сегодня день, потом посмотрит по таблице, какая сегодня скидка.
Получается, что в разные дни функция получает на вход 1000 ₽, но возвращает разные значения — так работает императивное программирование, когда всё зависит от других переменных.
Последовательность выполнения подпрограмм регулируется программистом. Он задаёт нужные условия, по которым движется программа. Вся логика полностью продумывается программистом — как он скажет, так и будет. Это значит, что разработчик может точно предсказать, в какой момент какой кусок кода выполнится — код получается предсказуемым, с понятной логикой работы.

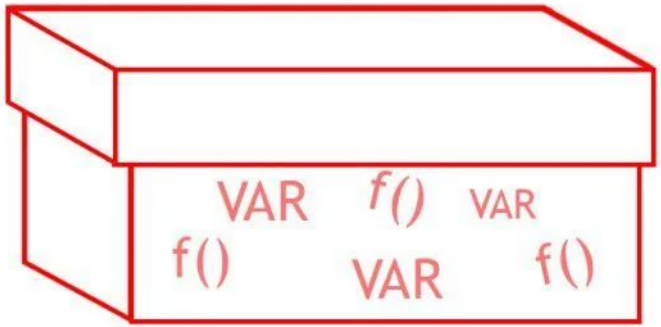
Функциональное программирование устроено так:

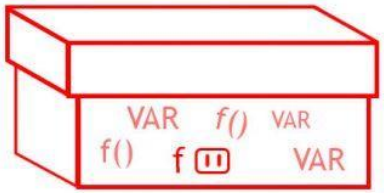
Мы задаём не последовательность нужных нам команд, а описываем взаимодействие между ними и подпрограммами. В функциональном программировании весь код — это правила работы с данными. Вы просто задаёте нужные правила, а код сам разбирается, как их применять.
Если мы сравним принципы функционального подхода с императивным, то единственное, что совпадёт, — и там, и там есть команды, которые язык может выполнять. Всё остальное — разное.
Команды можно собирать в подпрограммы, но их последовательность не имеет значения. Нет разницы, в каком порядке вы напишете подпрограммы — это же просто правила, а правила применяются тогда, когда нужно, а не когда про них сказали.
Переменных в том виде, в котором мы привыкли, нет. В функциональном языке мы можем объявить переменную только один раз, и после этого значение переменной измениться не может. Сами же промежуточные результаты хранятся в функциях — обратившись к нужной, вы всегда получите искомый результат.
Функции всегда возвращают одно и то же значение, если на вход поступают одни и те же данные.
Можно провести аналогию с математикой и синусами: синус 90 градусов всегда равен единице, в какой бы момент мы его ни посчитали или какие бы углы у нас ещё ни были в задаче. То же самое и здесь — всё предсказуемо и зависит только от входных параметров.
Последовательность выполнения подпрограмм определяет сам код и компилятор, а не программист. Каждая команда — это какое-то правило, поэтому нет разницы, когда мы запишем это правило, в начале или в конце кода. Главное, чтобы у нас это правило было, а компилятор сам разберётся, в какой момент его применять.

Мотивация ООП

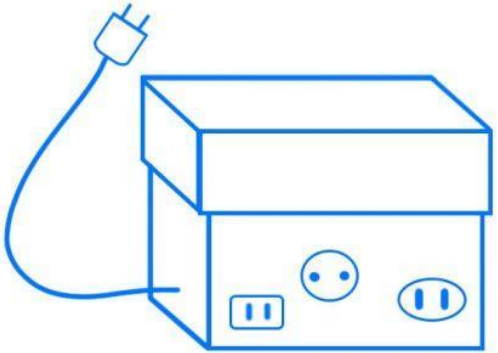


Это объект
Внутри лежат функции
и переменные

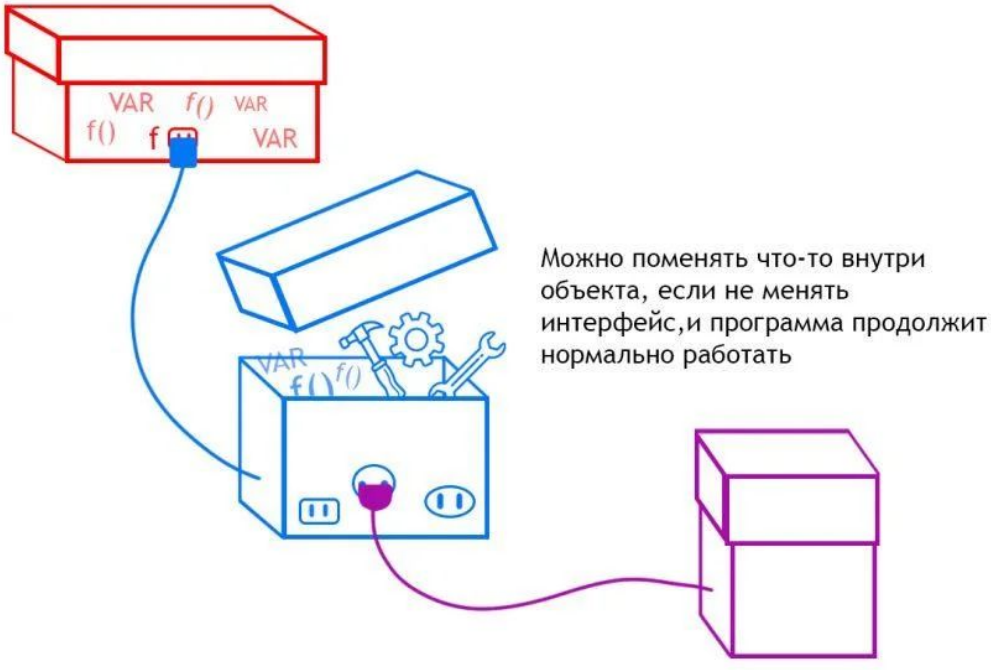




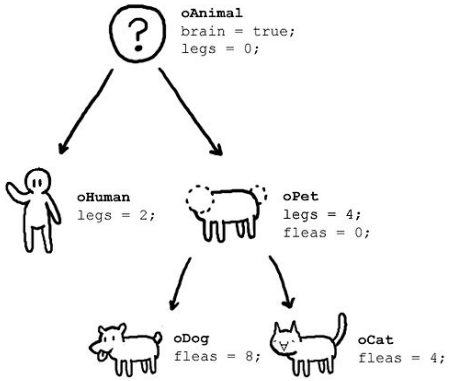
Это как бы интерфейс объекта
Я знаю: что бы в объекте ни произошло, этот интерфейс будет работать всегда одинаково



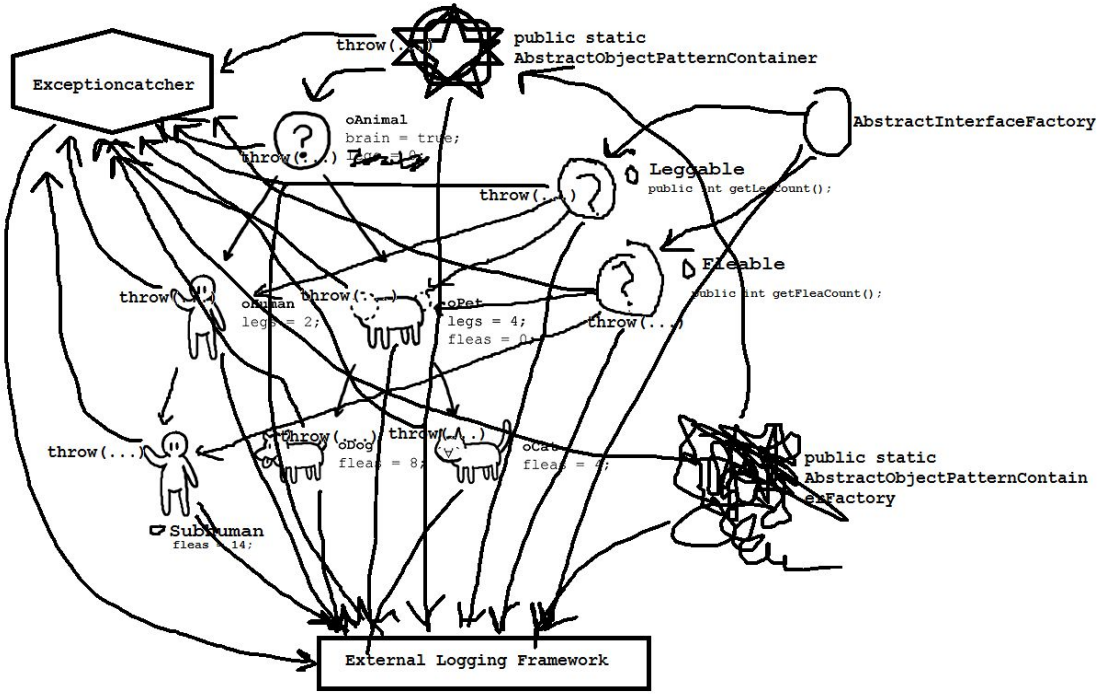
У других объектов могут быть свои интерфейсы


VAR f() VAR
f()
VAR
VAR
f()
f()
Можно поменять что-то внутри объекта, если не менять интерфейс,и программа продолжит нормально работать

What OOP users claim



What actually happens



Три парадигмы

текст

Инкапсуляция

Инкапсуляция — объект независим: каждый объект устроен так, что нужные для него данные живут внутри этого объекта, а не где-то снаружи в программе. Например, если у меня есть объект «Пользователь», то у меня в нём будут все данные о пользователе: и имя, и адрес, и всё остальное. И в нём же будут методы «Проверить адрес» или «Подписать на рассылку».
Абстракция — у объекта есть «интерфейс»: у объекта есть методы и свойства, к которым мы можем обратиться извне этого объекта. Так же, как мы можем нажать кнопку на блендере. У блендера есть много всего внутри, что заставляет его работать, но на главной панели есть только кнопка. Вот эта кнопка и есть абстрактный интерфейс.
В программе мы можем сказать: «Удалить пользователя». На языке ООП это будет «пользователь.удалить ()» — то есть мы обращаемся к объекту «пользователь» и вызываем метод «удалить». Кайф в том, что нам не так важно, как именно будет происходить удаление: ООП позволяет нам не думать об этом в момент обращения.
Например, над магазином работают два программиста: один пишет модуль заказа, а второй — модуль доставки. У первого в объекте «заказ» есть метод «отменить». И вот второму нужно из-за доставки отменить заказ. И он спокойно пишет: «заказ.отменить ()». Ему неважно, как другой программист будет реализовывать отмену: какие он отправит письма, что запишет в базу данных, какие выведет предупреждения.

Полиморфизм

Полиморфизм — единый язык общения. В ООП важно, чтобы все объекты общались друг с другом на понятном им языке. И если у разных объектов есть метод «Удалить», то он должен делать именно это и писаться везде одинаково. Нельзя, чтобы у одного объекта это было «Удалить», а у другого «Стереть».
При этом внутри объекта методы могут быть реализованы по-разному. Например, удалить товар — это выдать предупреждение, а потом пометить товар в базе данных как удалённый. А удалить пользователя — это отменить его покупки, отписать от рассылки и заархивировать историю его покупок. События разные, но для программиста это неважно. У него просто есть метод «Удалить ()», и он ему доверяет.

Наследование

Наследование — способность к копированию. ООП позволяет создавать много объектов по образу и подобию другого объекта. Это позволяет не копипастить код по двести раз, а один раз нормально написать и потом много раз использовать.
Например, у вас может быть некий идеальный объект «Пользователь»: в нём вы прописываете всё, что может происходить с пользователем. У вас могут быть свойства: имя, возраст, адрес, номер карты. И могут быть методы «Дать скидку», «Проверить заказ», «Найти заказы», «Позвонить».
На основе этого идеального пользователя вы можете создать реального «Покупателя Ивана». У него при создании будут все свойства и методы, которые вы задали у идеального покупателя, плюс могут быть какие-то свои, если захотите.

В Питоне

Пример простого класса Person с конструктором (def __init__), принимающим имя (name) и выводящим поля класса в определенном формате с помощью метода display_info. В __init__ вы должны задавать все поля, которые планируют быть в вашем классе.

```python
class Person:

    def __init__(self, name):
        self.name = name     # имя человека
        self.age = 1         # возраст человека

    def display_info(self):
        print(f"Name: {self.name}  Age: {self.age}")
```

Перехват исключений

```
f = open('1.txt')
ints = []
try:
    for line in f:
        ints.append(int(line))
except ValueError:
    print('Это не число. Выходим.')
except Exception:
    print('Это что ещё такое?')
else:
    print('Всё хорошо.')
finally:
    f.close()
    print('Я закрыл файл.')
```

Assert

```
def avg(ranks):
    assert len(ranks) != 0
    return round(sum(ranks)/len(ranks), 2)

ranks = [62, 65, 75]
print("Среднее значение:", avg(ranks))
```

```
ranks = []
print("Среднее значение:", avg(ranks))
```

```
def avg(ranks):
    assert len(ranks) != 0, 'Список ranks не должен быть пустым'
    return round(sum(ranks)/len(ranks), 2)

ranks = []
print("Среднее значение:", avg(ranks))
```

Интересные ссылки

https://otus.ru/journal/imperativnoe-i-deklarativnoe-programmirovanie/