

# Объектно-ориентированное программирование




Урок 3

Курс “Парадигмы программирования и языки парадигм”





## Цели семинара

-  Понять основные отличия между **ООП** и уже известными парадигмами
-  Научиться принимать решение об использовании **ООП** в конкретной задаче
-  Научиться решать задачи в рамках **ООП** парадигмы



## План семинара



Викторина



Пишем код



Решаем кейс



Подведение итогов



# Викторина





## Регламент

- 1 Прочитать код
- 2 Подумать в какой парадигме написана программа и поделиться своим ответом
- 3 Обсудить решение



## Что за парадигма: числа

```
1 numbers = [1, 2, 3, 4, 5]
2 squared_numbers = [x ** 2 for x in numbers]
3 even_numbers = [x for x in numbers if x % 2 == 0]
4 doubled_numbers = list(map(lambda x: x * 2, numbers))
5
6 print("Квадраты чисел:", squared_numbers)
7 print("Четные числа:", even_numbers)
8 print("Удвоенные числа:", doubled_numbers)
```

Ответ: ..?



## Что за парадигма: числа

```
1 numbers = [1, 2, 3, 4, 5]
2 squared_numbers = [x ** 2 for x in numbers]
3 even_numbers = [x for x in numbers if x % 2 == 0]
4 doubled_numbers = list(map(lambda x: x * 2, numbers))
5
6 print("Квадраты чисел:", squared_numbers)
7 print("Четные числа:", even_numbers)
8 print("Удвоенные числа:", doubled_numbers)
```

**Ответ:** Используются *императивная* и *декларативная* парадигмы!

**Почему это так:** генераторы списков и метод `map` - признаки декларативной (если быть точным - функциональной) парадигмы, а весь остальной код остаётся в рамках императивного стиля



## Что за парадигма: автомобиль

```
1 class Vehicle:
2     def __init__(self, brand, color):
3         self.brand = brand
4         self.color = color
5
6 class Car(Vehicle):
7     def start_engine(self):
8         print(f"{self.color} автомобиль {self.brand} завел
9             двигатель.")
10
11     def stop_engine(self):
12         print(f"{self.color} автомобиль {self.brand} заглушил
13             двигатель.")
```

Ответ: ..?





## Что за парадигма: автомобиль

```
1 class Vehicle:
2     def __init__(self, brand, color):
3         self.brand = brand
4         self.color = color
5
6 class Car(Vehicle):
7     def start_engine(self):
8         print(f"{self.color} автомобиль {self.brand} завел
9         двигатель.")
10
11     def stop_engine(self):
12         print(f"{self.color} автомобиль {self.brand} заглушил
13         двигатель.")
```

**Ответ:** это *объектно-ориентированная парадигма*.

**Почему это так:** в данном коде определяются два **класса** *Vehicle* и *Car* и их **методы** `__init__`, `start_engine`, `stop_engine`.



## Что за парадигма: музыкальный плейлист

```
1 class MusicPlayer:
2     def __init__(self):
3         self.playlist = []
4
5     def add_song(self, song):
6         self.playlist.append(song)
7
8 def create_playlist(player: MusicPlayer, songs: list):
9     for song in songs:
10         player.add_song(song)
11
12 player = MusicPlayer()
13 create_playlist(player)
```

Ответ: ..?



## Что за парадигма: музыкальный плейлист

```
1 class MusicPlayer:
2     def __init__(self):
3         self.playlist = []
4
5     def add_song(self, song):
6         self.playlist.append(song)
7
8 def create_playlist(player:MusicPlayer, songs:list):
9     for song in songs:
10         player.add_song(song)
11
12 player = MusicPlayer()
13 create_playlist(player)
```

**Ответ:** здесь использованы *ООП*, *процедурная* и *структурная* парадигмы.

**Почему это так:** объявлен класс *MusicPlayer*, его конструктор и метод *add\_song*. Для создания плейлиста используется стандартная процедура (НЕ метод), также используется цикл *for*, и нет *goto*.



# Вопросы



Пишем код





## Регламент

- 1 Вместе читаем условия задачи
- 2 Вы самостоятельно решаете задачу
- 3 Вместе обсуждаем решение



# Геометрические фигуры: абстрактный класс



## Геометрические фигуры: абстрактный класс

- **Контекст**  
Предположим, что мы хотим написать программу для исследования геометрических фигур. Для того чтобы это сделать мы решили начать с создания абстрактного класса - “Фигура”.
- **Задача**  
Реализовать класс *Shape*, содержащий пустые методы *get\_area* и *get\_perimeter*. Использовать библиотеку абстрактных классов “ABC” в данном случае - не обязательно.
- **Решение.. ?**





## Геометрические фигуры: абстрактный класс

- **Контекст**

Предположим, что мы хотим написать программу для исследования геометрических фигур. Для того чтобы это сделать мы решили начать с создания абстрактного класса - “Фигура”.

- **Задача**

Реализовать класс *Shape*, содержащий пустые методы *get\_area* и *get\_perimeter*. Использовать библиотеку абстрактных классов “ABC” в данном случае - не обязательно.

- **Решение:**

```
1 class Shape:
2     def get_perimeter(self):
3         pass
4
5     def get_area(self):
6         pass
```



# Геометрические фигуры: круг



## Геометрические фигуры: круг

- **Контекст**

Теперь, когда у вас есть абстрактный класс *Shape*, ваша следующая задача - получить класс *Circle*.

- **Задача**

Реализовать дочерний от *Shape* класс *Circle*, включая следующие работающие методы:

- конструктор класса `__init__` - метод инициализации класса *Circle*.
- `get_area` - метод для расчета площади круга
- `get_perimeter` - метод для расчета периметра окружности

- **Решение.. ?**



## Геометрические фигуры: круг

- **Задача**

Реализовать дочерний от *Shape* класс *Circle*, включая следующие работающие методы:

- конструктор класса `__init__` - метод инициализации класса *Circle*.
- `get_area` - метод для расчета площади круга
- `get_perimeter` - метод для расчета периметра окружности

- **Решение:**

```
1 import math
2
3 class Circle(Shape):
4     def __init__(self, radius):
5         self.radius = radius
6
7     def get_perimeter(self):
8         return 2 * math.pi * self.radius
9
10    def get_area(self):
11        return math.pi * self.radius ** 2
```



# Геометрические фигуры: треугольник



## Геометрические фигуры: треугольник

- **Контекст**

И наконец, последняя задача - по аналогии с кругом создать класс для треугольника и расчета его характеристик.

- **Задача**

Реализовать дочерний от *Shape* класс *Triangle*, включая следующие работающие методы:

- конструктор класса `__init__` - метод инициализации класса.
- `get_area` - метод для расчета площади
- `get_perimeter` - метод для расчета периметра

- **Решение.. ?**



## Геометрические фигуры: треугольник

- Решение:

```
1 import math
2
3 class Triangle(Shape):
4     def __init__(self, a, b, c):
5         self.a = a
6         self.b = b
7         self.c = c
8
9     def get_perimeter(self):
10        return self.a + self.b + self.c
11
12    def get_area(self):
13        p = self.get_perimeter() / 2
14        return math.sqrt(p * (p - self.a) * (p - self.b) * (p -
        self.c))
```



Решаем кейс







## Регламент

- 1 Вместе читаем кейс
- 2 Вы думаете как можно его решить
- 3 Вместе обсуждаем решение



# Робот-пылесос

## Кейс. Робот-пылесос

### Контекст:

Вас наняли на работу в компанию производящей бытовую технику, основной продукт которой - робот-пылесос. Ваша команда занимается разработкой “головы” пылесоса, то есть таким функционалом как: планирование пути, локализация, управление роботом, обработка сенсоров.

**Робот пылесос** - это устройство бытовой электроники, которое автоматизирует процесс пылесосной уборки вашего дома. Как правило оно реализовано в виде пластикового круглого робота на 2-4 колесах, которое само ездит по вашей комнате, причём так, чтобы максимизировать площадь покрытия комнаты / квартиры.

**ТЗ:** Разработать модуль управления роботом, с помощью которого приводится в исполнение тактика взаимодействия робота с окружающим миром. Получаем на вход некоторый набор переменных и на их основе приводим робота в движение определённым образом.

**Задача:** Какие парадигмы вы будете использовать для разработки такого ПО и почему именно их?



## Кейс. Робот-пылесос

### Обсуждение:

Попробуем примерно представить как верхнеуровнево выглядел бы такой модуль управления. Первое что приходит в голову - есть какой-то обработчик событий, то есть *бесконечный цикл*, который обрывается если получит *сигнал отмены*, и внутри которого происходят некоторые *события* и их обработка. Скорее всего и структурная и процедурная парадигмы - здесь будут использованы для описание такого цикла.

Также сразу видно что речь идёт об описании сложных объектов и их взаимодействий, например: у пылесоса есть составляющие (колеса, щётки, корпус и прочие). У этих составляющих могут быть состояния (колеса вращаются или стоят на месте, корпус повернут передом в одну из сторон), состояния могут меняться в зависимости от полученных в каждый момент времени данных. Похоже, что в данном случае не обойтись без ООП.

Про декларативную парадигму здесь можно поспорить, но в общем виде ответ - **не понятно** без технологического стека нашей компании.





# Итоги семинара





## Итоги семинара



### Викторина “Что за парадигма”

- Решили 3 задачи на классификацию парадигм



### Пишем код

- Решили 3 задачи по программированию



### Решаем кейсы

- Решили и обсудили кейс “Робот-пылесос”



### Подвели итоги



# Домашнее задание



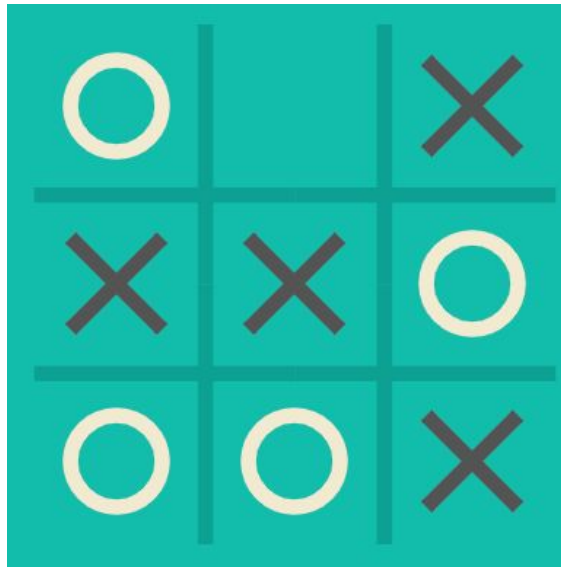
## Крестики-нолики

- **Контекст**

Вероятнее всего, вы с детства знакомы с этой игрой. Пришло время реализовать её. Два игрока по очереди ставят крестики и нолики на игровое поле. Игра завершается когда кто-то победил, либо наступила ничья, либо игроки отказались играть.

- **Задача**

Написать игру в “Крестики-нолики”. Можете использовать любые парадигмы, которые посчитаете наиболее подходящими. Можете реализовать доску как угодно - как одномерный массив или двумерный массив (массив массивов). Можете использовать как правила, так и хардкод, на своё усмотрение. Главное, чтобы в игру можно было поиграть через терминал с вашего компьютера.







Конец семинара  
Спасибо за внимание!

