GeekBrains

Объектно-ориентированное программирование (лекция 3)

Парадигмы программирования и языки парадигм

Оглавление

Вступление

Добрый день, коллеги! Меня зовут Александр Левин. В сегодняшней лекции мы погрузимся в парадигму объектно-ориентированного программирования, также известную под сокращенным названием - ООП.

План на сегодня примерно следующий:

- Мы начнём с описания контекста вокруг ООП.
- Затем погрузимся в суть определения и попробуем привести примеры: **объектов**, их **атрибутов** и взаимосвязи между ними, для того чтобы более детально понять суть парадигмы.
- После этого более подробно рассмотрим 4 концепции ООП.
- Потом, мы кратко разберем историю этого термина: как появилось объектно-ориентированное программирование (сокращённо - ООП) и главное зачем (то есть, какие проблемы решает данный подход).
- Далее, рассмотрим примеры ООП: сначала на пальцах, потом в коде, а также перечислим несколько ООП языков.
- Наконец, обсудим преимущества и недостатки данной парадигмы.

Данная лекция призвана сформировать представление о том, что такое ООП и в каких случаях его можно и стоит применять. Если после окончания лекции у вас останутся вопросы, вы сможете написать их мне в Telegram, а контакт я оставлю в конце лекции.

Начнем с общего описания контекста применения объектно-ориентированной парадигмы.

Термины, используемые в лекции

Объектно-ориентированное программирование – определение.

Класс – определение.

Объект – определение.

Атрибут – определение.

Метод – определение.

Наследование – определение

Инкапсуляция – определение

Полиморфизм – определение

Абстракция – определение

Контекст

ООП на сегодняшней день является крайне важной и к тому же широко распространенной парадигмой. Она встречается в различных прикладных областях, являясь особенно востребованной в тех кейсах, когда сам проект сложен и/или будет масштабирован. Вот лишь некоторые примеры проектов, где без ООП никак:

- Разработка игр и симуляций
- Разработка ПО: браузеры, антивирусы, графические редакторы, текстовые процессоры, ваш любимый мессенджер или VPN
- Разработка мобильных приложений: как на iOS, так и на Android, все мобильные приложения строятся на принципах ООП
- Разработка веб-приложений
- Разработка систем управления базами данных (СУБД)
- Разработка фреймворков и библиотек программирования
- Разработка сложных enterprise приложений: например, ERP, CRM, CMS, автоматизации бизнес-процессов и множества других

В подобных проектах сложность с течением времени разработанного ПО быстро возрастает. ООП предоставляет гибкость, а также упрощает поддержку и сопровождение программ. Давайте попробуем разобраться в чем именно заключается это удобство.

В парадигмах из предыдущей лекции акцент ставился на том “что именно и как делать”. В процедурной парадигме у самих процедур были понятные “входы” и “выходы”. В структурном были управляющие блоки, с помощью которых можно реализовать любой алгоритм. При этом, нам не было интересно ни то какие связи существуют между взаимодействующими переменными внутри одной операции. Ни то как хранятся и хранятся ли вообще промежуточные результаты. А в ООП все наоборот: вначале нам может быть вообще не интересно что мы будем делать с объектами в будущем и какими будут входы. Вместо этого, сначала мы выстраиваем некую структуру объекта, называемую классом: указываем что это за объект, что содержится внутри и что с этим МОЖНО сделать в теории в будущем (объекты создаются из классов и имеют атрибуты, а в атрибутах можно хранить любую информацию). Таким образом, особенность ООП в акценте на объектах и взаимодействиях между ними.

Что такое ООП

Предлагаю перейти к более строгим определениям и разобрать их поподробнее.

Определения

Также как и в предыдущих парадигмах (структурной и процедурной), ООП отличается от чисто императивного подхода некоторыми новыми концепциями и возможностями, которые даёт реализация этих новых концепций уже в языках программирования. Возможно вы уже слышали такие термины, как например: **объект**, **класс**, **метод** класса, **атрибут** класса, **экземпляр** класса, **родительский** класс, **дочерний** класс - все это термины из объектно-ориентированной парадигмы. Сейчас мы с вами разберем эти страшные (а может быть и разжигающие любопытство) слова.

Начнем с самой парадигмы. **ООП** - это императивная парадигма программирования, основанная на представлении кода в виде множества объектов и последовательности их взаимодействий. Таким образом, основной фундамент ООП - это идея **классов** и **объектов**. **Класс** - это тип объекта, то есть шаблон по которому можно создать объект (проще говоря, чертеж). А **Объект** - это экземпляр класса. Например, “модель автомобиля” - это **класс**, а конкретный отдельно взятый автомобиль - это **объект**.

Говоря прикладным языком, чтобы создать объект - обязательно заранее нужно иметь инструкцию о том, что точно должно быть внутри. Такая инструкция и называется классом. По факту, объект это что-то что лежит в оперативной памяти или на диске, а класс - это программный код или другая инструкция, с помощью которой можно создать объект. Как правило, когда вы создаёте объект, вы уже заранее знаете какого класса он будет.

Каждый объект имеет свой **класс** (то есть **тип**), из которого он был создан. Пускай мы имеем объект типа “X”. Если предположить, что класс “X” был преобразован в процессе исполнения программы, то класс самого объекта типа “X” останется неизменным и будет равным “X”. С другой стороны, если на основе класса “X” будет создан другой класс “Y”, то все объекты полученные из класса “Y” будут иметь тип “Y”. Этот принцип называется **наследованием** и будет разобран позже.

Теперь заглянем внутрь объектов. Данные содержащиеся в объектах называются **атрибутами** или **свойствами**, а код подпрограмм - **методами** данного объекта. **Атрибуты** - это точно такие же переменные, как в любой императивной программе. а **методы** - это точно такие же функции или процедуры. Единственное отличие в том, что здесь они все принадлежат своим объектам.

Что значит **принадлежат**? Во-первых, вы можете получить конкретный атрибут только через его объект. Это похоже на процесс открытия файла из какой-нибудь папки в операционной системе. Сначала вы должны пройти путь до папки где лежит ваш файл, и только после этого сможете его открыть. Если вы не знаете где лежит ваш файл, то открыть его не получится.

Например, давайте попробуем открыть библиотеку pandas в языке Python. Для этого напишем строку с импортом модуля и распечатаем результат на экран.

```
1 import pandas as pd
2 print(pd)
```

В языке Python - все является объектом, а если сделать print(object), мы увидим что это за объект с некоторыми подробностями. Как мы видим, объект pd - это **модуль**, который называется pandas и лежит по указанному пути.
Output:

```
<module 'pandas' from 'c:\\Users\\
```

В этом модуле есть специальный класс для работы с табличными данными, который называется DataFrame, давайте вызовем его.

```
1 import pandas as pd
2 print(pd.DataFrame)
```

Для указания принадлежности объекту, в Python используется точка, указываемая после этого объекта. Отлично, Python сообщает нам что мы смотрим на объект типа **class**, называемый pandas.core.frame.DataFrame.
Output:

```
<class 'pandas.core.frame.DataFrame'>
```

Давайте теперь получим метод describe из объекта DataFrame.

```
1 import pandas as pd
2 print(pd.DataFrame.describe)
```

И вот мы уже смотрим на объект типа **function**, который называется NDFrame.describe, и он даже лежит в какой-то своей ячейке памяти.

Output:

```
<function NDFrame.describe at 0x000001AB2E30F9D0>
```

Таким образом, мы видим, что метод NDFrame.describe принадлежит объекту DataFrame, который в свою очередь принадлежит модулю pandas. То есть принадлежность определяет расположение объекта. В Python ещё есть прекрасный встроенный метод **dir**, который делает ровно то же, что и в командной строке Windows: выводит все объекты, содержащиеся в текущей папке (то есть, в текущем объекте):

```
1 dir(pd.DataFrame)
```

Рекомендую повторить это у себя самостоятельно.

Во-вторых, это означает, что метод или атрибут является локальным для всего что происходит внутри объекта (находится в **локальной области видимости**) и, по умолчанию, не виден для всего что происходит за его пределами. Этот замечательный принцип - называется **инкапсуляцией**, и он будет разобран подробнее дальше.

Свойства объектов и классов

Чтобы глубже понять тонкости ООП, предлагаю разобрать некоторые свойства классов. Я буду приводить высказывания, а потом мы будем разбирать их поподробнее.

- Любой объект имеет **тип** - то есть класс, по инструкции которого был получен данный объект.

То есть не бывает объектов без типа, каждый объект обязательно наследует какому-то классу. Как правило, в языках программирования существует специальный синтаксис для определения типа объекта. Например, в Python данная операция реализована во встроенном методе type, а в Java с помощью метода getClass.

- Любой класс имеет свой родительский класс, то есть такой класс которому наследует этот класс.

В Python например все упирается в класс type - это одновременно и метод и такой метакласс, из которого вытекают все остальные классы. **Метакласс** - это такой класс, который создает классы.

- Атрибутом класса может быть все что угодно, то есть объект любого типа.

Как мы уже знаем, атрибут - это просто переменная, поэтому все то что можно положить в переменную, по умолчанию можно положить и в класс

- Методом класса может быть только функция или процедура.

Так как метод - это функция или процедура принадлежащая классу, по определению.

И напоследок, предлагаю ещё раз повторить важные определения:
И так,

- **Класс** - это тип объекта
- **Объект** - это экземпляр класса (контейнер для данных и методов)
- **Метод** - это процедура или функция, принадлежащая классу
- **Атрибут** - это единица данных, принадлежащая классу.
- **Программа в ООП** - это описание взаимодействий объектов.
- **Объектно-ориентированное программирование** - это императивная парадигма программирования, основанная на представлении кода в виде множества объектов и последовательности их взаимодействий

В реальной жизни процесс разработки на ООП может выглядеть следующим образом. Сначала, вы определяете какие примерно классы вам нужны, их названия и примерно что они будут из себя представлять. Затем вы можете погрузиться на уровень ниже и уточнить что будет храниться в каждом классе. Ну а когда вы уже знаете “что это такое” и “что внутри”, остается только написать методы, то есть “что с этим можно сделать”.

Концепции ООП

Предлагаю теперь более детально обсудить важнейшие концепции парадигмы ООП.

Инкапсуляция

Первый принцип, который мы рассмотрим, называется “**инкапсуляция**” (англ. - encapsulation). Слово “**инкапсуляция**” можно перевести как помещение чего-то в “капсулу”, то есть в какое-то изолированное пространство внутри исходного. В программировании под инкапсуляцией понимается разделение интерфейса

какого-то функционала системы (например, программного обеспечения) от его реализации. **Интерфейс** в данном контексте - это множество инструментов пользователя для взаимодействия с системой. То есть, говоря простыми словами, есть то, что система из себя представляет изнутри, а есть то, что видит пользователь. Вот интерфейс - это как раз то, что видит пользователь. А вот такое разделение на внешнюю и внутреннюю часть, когда внутренности частично или полностью скрыты от пользователя - и есть **инкапсуляция**.

Простейший пример инкапсуляции - железо внутри корпуса компьютера. Неважно ноутбук у вас или системный блок: у вас, как у пользователя, скорее всего сходу нет доступа к его внутренностям. Как правило, для того чтобы его получить и увидеть как компьютер устроен на уровне печатных плат, необходимо как минимум разобрать корпус. С другой стороны, у вас есть удобный интерфейс в виде клавиатуры, мыши, экрана, и других кнопок на корпусе для того чтобы с компьютером успешно взаимодействовать.

Но вернемся к программированию. В ООП, инкапсуляция в первую очередь необходима для обеспечения безопасности данных и методов внутри объектов. Возможно вы задались вопросом: “Безопасность от чего?”. Ответ - от случайных ошибок и непредвиденных изменений в тех местах, где это делать не планировалось. Если вернутся к нашему примеру с компьютером, то было бы очень неловко случайно отсоединить какой-нибудь важный провод в попытке просто его включить. Если бы инкапсуляции не было, такое легко могло бы происходить. Поэтому корпус защищает от таких взаимодействий, которые могут навредить системе.

Второе, для чего нужна инкапсуляция - это просто для удобства использования. Скорее всего, вам не нужно каждый раз перевтыкать проводки между платами внутри вашего ПК для того чтобы просто запустить его или сделать звук погромче. Все эти внутренности вам как пользователю чаще всего будут только мазолить глаза. С программированием все то же самое: вам как программисту для использования каких-либо объектов, далеко не всегда нужно залезать внутрь этого объекта (например, в код класса), чтобы что-то исправить. Чаще всего вы инициализируете этот объект для конкретного случая и просто используете его. Соответственно, неудобно было бы смотреть на большую часть того что происходит внутри когда вы это делаете. Так, инкапсуляция делает ваш код не только безопасным, но и красивым.

Наследование

Второй принцип называется **наследованием**. Смысл наследования довольно близок к интуитивному восприятию этого слова. **Наследование** (англ. - inheritance) - это концепция, согласно которой какой-то класс может получать свойства другого класса в момент своей инициализации. В таком случае новый класс называется “дочерним” или “подклассом”, а тот чьи свойства были скопированы - “родительским” или “суперклассом”.

💡 Копируются при наследовании как методы, так и атрибуты

Первое преимущество, которое дает данный принцип - это освобождение от **дублирования кода** в разных классах. Например, если вы уже имеете класс “чашка”, то вам не нужно создавать класс “большая красная чашка” с нуля. Поскольку “большая красная чашка” - это такая “чашка”, которая “большая” и “красная”, то вам достаточно просто унаследовать все свойства чашки и дополнительно просто прописать то, что она ещё “большая” и “красная”.

Второе преимущество, которое дает наследование - это **упрощение поддержки и изменений** программ. Предположим, что у вас есть код, в котором классы выстраиваются в сложную цепочку наследований, и вы вдруг захотели поменять что-то в классе достаточно высокого уровня, например, у которого есть и классы-дети и классы-правнуки. В этом случае вам достаточно будет произвести изменение только в одном месте. После того, как вы измените то что вам нужно в этом высоком родительском классе, все дочерние просто унаследуют данное изменение, а вам не придется менять каждый из них по отдельности.

Полиморфизм

Следующий принцип называется “**полиморфизм**”. Полиморфизм переводится с греческого как “множество форм” или “многоформенный”. Приведу несколько определений полиморфизма, после чего разберем их поподробнее. И так:

- **Полиморфизм** - это механизм предоставления единого интерфейса взаимодействия к объектам различных типов.
- **Полиморфизм** - это возможность обработки различных типов данных с помощью единого функционала
- **Полиморфизм** - это семейство различных механизмов, позволяющих использовать один и тот же участок программы с различными типами в различных контекстах.

Суть всех определений одна: есть какой-то код (например, метода класса), у него есть какие-то аргументы, и ещё есть множество контекстов в которых этот метод

можно использовать. В разных контекстах, типы входных данных для этого метода могут отличаться. Например, операция сложения может быть определена для чисел, а может быть определена для строк. В Python, например, сложение для строк производит их конкатенацию (склеивание строк). Так вот, если у вас есть возможность при разработке использовать один и тот же метод с разными типами данных не называя его по-другому, а вызывая именно тем же именем (или операндом), значит у вас есть возможность использовать полиморфизм.

Как этим пользоваться? Полиморфизм - это концепция, которую можно реализовывать разными способами. Один из распространенных способов реализации полиморфизма называется **переопределением** (англ. - overloading) или **перегрузкой**, если переводить буквально (именно такой термин встречается чаще, однако по-русски "**переопределение**" более точно отражает его суть). **Переопределение** - это такая реализация полиморфизма, при которой оператор переопределяется для нескольких типов данных в одной области видимости. Разберём небольшой пример на Python. Предположим у нас есть классы Parent и Child:

```
1 class Parent:
2     def execute(self):
3         print('Executing a Parent class')
4 
5 class Child(Parent):
6     def execute(self):
7         print('Executing a Child class')
```

Parent - это класс, у которого есть метод *execute*. Child - это класс, который наследует классу *Parent*, и у него тоже есть метод execute, НО: в классе *Child* метод execute переопределён. Таким образом, если мы исполняем метод *execute* на объекте типа Parent, то исполнится execute из класса *Parent*, и будет выведена надпись "Exectuing a Parent class". А если на объекте типа *Child*, то выведется, соответственно, "Executing a Child class". Конечно данный пример не является хоть сколько-нибудь полезным, но он отлично демонстрирует полиморфизм в Python на практике. Самое главное здесь то, что если мы хотим исполнить *execute*, то нам не нужно думать о том какой тип объекта мы используем (Parent или Child), так как у них обоих есть соответствующий метод, который работает именно для их типа.

Повторим ещё раз простыми словами: полиморфизм позволяет обрабатывать разные типы данных с помощью одного интерфейса. То есть, типов данных много, а интерфейс один.

Абстракция

Последний принцип, который мы рассмотрим называтеся “абстракцией”. Слово абстракция (англ. - abstraction, лат. - abstractio) лучше всего переводится как “отделение” или “отвлечение”. Давайте на секунду *абстрагируемся* от объектов и программирования.

Абстракция - это метод научного познания, основанный на выделении “важных” признаков объекта познания и отделения их от “неважных”. Но помимо этого, **абстракцией** также называют результат этого метода, то есть сам объект содержащий только важные признаки и не содержащий неважные.

Получается так: предположим был какой-то объект представляющий собой совокупность свойств. Нам для его изучения в конкретном кейсе часть свойств не понадобилась, и чтобы не отвлекаться на них, мы их просто отбросили. То, что осталось - называют **абстракцией**. НО и сам процесс тоже называют **абстракцией**. Важно понимать что имеется в виду под данным словом в зависимости от контекста: объект или процесс его получения.

Теперь предлагаю вернуться к ООП. В ООП абстракция означает нечто очень похожее. Если программа в ООП - это описание взаимодействий объектов, значит эти объекты необходимо предварительно описать. Для того чтобы описать объект, используется принцип **абстракции**, следуя которому мы имеем право описывать только те свойства объекта, которые для нас важны, игнорируя при этом тот факт, что вообще-то в реальной жизни этот объект устроен намного сложнее. Под важностью в данном случае понимается использование данного признака в вычислениях: если вы не хотите его использовать, значит и учитывать его не обязательно. Поэтому хорошо бы заранее хотя бы примерно понимать как именно вы хотите использовать тот объект, который сейчас создаете.

Разберем абстракцию на примере. Пусть у вас есть задача про баскетбол. Например, по силе и вектору броска определить траекторию движения мяча в сторону корзины. От чего будет зависеть траектория? Точно от тех вводных, которые мы уже учли: “силы” и “вектора броска”. А ещё от “массы” и “размера мяча”, “плотности материала” и “силы ветра”. Наверняка, такие признаки вы бы сочли важными для решения данной задачи. Но есть и другие признаки описывающие взаимодействия объектов в нашем примере, такие как “цвет мяча”, “тип спортивной одежды бросающего”, “геопозиция”, “текущий день недели”, которыми в данной ситуации явно можно пренебречь.

Процесс построения таких объектов “мяча” и “бросающего”, которые бы содержали только важные атрибуты называется абстракцией. Но после того, как вы описали этот мяч и нападающего с важными свойствами и без неважных - они тоже называются абстракцией.

Таким образом, абстракция позволяет очень сильно упростить задачу и для программиста и для компьютера, не учитывая те характеристики, которые не влияют или влияют на результат в очень малой степени.

История вкратце

Давайте вкратце изучим как появилось ООП.

Первым ООП языком считается Simula 67 созданная в 1965 году и выпущенная в 1967-м сотрудниками Норвежского вычислительного центра в Осло Кристеном Нюгором и Оле-Йоханом Далем для моделирования процессов в сложных системах, то есть разработки симуляций. Как раз это и следует из названия Simula. Например Simula использовался исследователями, занимающимися физическим моделированием, для изучения и улучшения движения судов и их содержимого через грузовые порты.

Так вот, разработка симуляций выделяется большим количеством переменных, состояния которых в каждый момент времени нужно где-то хранить. При этом хранить всё в одном большом массиве памяти (в общем котле) не очень удобно, т.к. уследить за объемным набором имён довольно сложно, зато легко запутаться и забыть что с чем связано. В то же время у вас есть процедурный подход, в рамках которого, как вы помните, программа представляет собой исключительно множество последовательно вызываемых процедур.

💡 **Процедура** - это инструкция, выделенная в отдельный блок кода, который можно вызывать по его имени, то есть имени процедуры и которая выполняется для конкретных входных значений.

Важнее всего, что при этом процедура не хранит в себе информацию о связях между переменными и характере этих связей. Чтобы устранить этот пробел, в Simula 67 были представлены такие концепты как класс и объект, наследование и динамическое связывание.

Первым «чистым» ООП языком считается язык Smalltalk, который представили в 1972 году сотрудники научно-исследовательского центра Xerox PARC Алан Кэй, Дэн

Инглз и другие в рамках проекта по созданию устройства Dynabook. Под «Чистым» ООП понимается язык, все типы которого являются или могут быть представлены классами. Также, в Smalltalk классы могут создаваться и изменяться динамически, а не только статически как в Simula (то есть тип объектов может изменяться в процессе исполнения).

Переходя к более современным языкам, Бьерн Страуструп разработал C++ как раз на основе привнесения концепций языка Simula в язык Си

💡 А ещё, потом он использовал Simula в свой кандидатской диссертации

Современные языки программирования
К современным языкам программирования поддерживающим ООП относятся:

- Python
- Java
- C++
- C#
- Ruby
- Swift
- Kotlin
- Objective-C

А к языкам без ООП:

- C
- Pascal
- Bash
- Fortran

Также, интересным примером является язык GO, который не считается ООП языком, так как там нет слова “объект”. Но де-факто он поддерживает большую часть принципов ООП, и таким образом может быть использован для разработки в ООП парадигме.

Примеры на пальцах

Предлагаю теперь разобрать примеры, чтобы разобраться в ООП на уровне практики.

Пример с возведением дома

Предположим мы хотим построить дом. Для этого у нас есть чертёж дома “Красивый красный дом” - это класс. Когда мы построим дом по этому чертежу - он станет объектом, назовём его “Построенный дом”, а его содержимое, например “окна” и “двери” - атрибутами. При этом наш дом будет иметь тип “Красивый красный дом”, т.к. именно по этому чертежу мы изначально его строили. То есть, у нашего объекта “Построенный дом” - класс (или ещё говорят тип) “Красивый красный дом”. Атрибутами этого класса может быть всё что мы захотим. Если в нашем чертеже не было камина и джакузи, не вопрос, можно добавить их после, но все атрибуты которые уже были заложены в чертеже (вроде окон и дверей) - точно появятся в объекте при его создании. Родительский класс нашего объекта всё равно останется прежним. Ещё важно уточнить, что атрибутами нашего класса могут быть как другие объекты, так и методы. Метод - это функция, которая является атрибутом объекта. В примере с домом методом может быть “открытие двери” или “включение электричества”.

Дети и родители

Предположим, у двух людей появился ребёнок. По сути, класс ребёнка полностью описан генами родителей и заложен при рождении. Но в процессе появления ребёнок довольно быстро перестаёт быть частью своих родителей и сам становится “объектом” :). При этом совершенно не исключено, что человек будет меняться с течением своей жизни и впоследствии станет сильно отличаться от своих родителей, но его генетическая принадлежность родителям всё равно никогда не изменится, т.к. он был создан ими в соответствии с их скомбинированной ДНК.

Но дальше происходит интересное: если всё идёт хорошо, человек в какой-то момент встречает кого-то и сам начинает создавать своих детей, то есть он сам становится классом. В программировании по счастливому стечению обстоятельств такой процесс называется наследованием. Наследование (англ. inheritance) - это создание нового класса на основе другого существующего класса. То есть впоследствии мы сможем создавать более узконаправленные классы для конкретных случаев, но использовать при этом один и тот же базовый (родительский) класс. Такой подход позволяет избежать дублирующегося кода, что в свою очередь конечно же сильно упрощает внесение правок в этот код и экономит время.

Давайте рассмотрим третий пример, чтобы увидеть удобство которое приносит нам наследование.

Разработка шутера

Предположим, что вам нужно разработать игру, например шутер от первого лица, в котором будет выбор внешнего вида персонажа. Чтобы сделать одного персонажа для игры вам потребуется приложить немало усилий: 3D-модель в виртуальном мире, анимация и скелет персонажа чтобы оживить его, текстуры внешнего вида, логика взаимодействий. А нам ведь всего-то хочется, чтобы внешний вид отличался. Все персонажи в нашей игре - гуманоиды, две ноги, две руки, ходят и выглядят примерно одинаково, взаимодействуют с миром тоже. Было бы здорово иметь какой-то базовый класс, содержащий все эти атрибуты, чтобы не пересоздавать их каждый раз от руки. Разработчики игр на самом деле довольно часто как раз и создают такой базовый класс, назовём его “персонаж”. На его основе можно уже создавать свой собственный объект, либо наследовать и создавать свой собственный класс, какой-нибудь “персонаж_в_бандане”. Кстати, игровые движки как раз полезны разработчикам игр тем, что экономят им кучу времени, предоставляя множество таких базовых классов для разных контекстов и ситуаций.

Очень важно в этом примере, что если мы захотим, чтобы все персонажи разом стали ниже или научились делать что-то новое, нам не нужно будет добавлять это изменение в каждый экземпляр этого класса. Достаточно будет изменить родительский класс и скомпилировать всё заново, чтобы объекты получили унаследованное изменение.

Давайте теперь обсудим конкретные ООП языки и посмотрим на код написанный на этих язык.

Примеры кода

Прямоугольник

Рассмотрим простой планиметрический объект - прямоугольник. Предположим, нам поставили задачу написать программу для расчета по его ширине и длине следующих его свойств:

- ☐ площади
- ☐ периметра
- ☐ длины диагонали
- ☐ углов между диагоналями

Начнем со словесного описания класса прямоугольника. Мы знаем, что это такой четырехугольный геометрический объект, у которого все углы прямые, то есть, равны 90 градусам. Таким образом, гарантируется, что его стороны попарно равны,

то есть он ещё и является параллелограммом. Для нас это означает, что объект “прямоугольник” возможно описать двумя числами: его длиной и шириной. Давайте сделаем это в Python. Если вы уже умеете использовать ООП в Python, попробуете прямо сейчас реализовать инициализацию “прямоугольника” самостоятельно.

```
1 class Rectangle:
2     pass
```

Для начала нам необходимо объявить Питону, что сейчас мы будем определять новый класс. Для того, чтобы это сделать здесь используется ключевое слово с говорящим названием: **class**. После ключевого слова **class** необходимо передать имя нового класса, мы назовем его просто Rectangle (то есть “прямоугольник”). Далее следует двоеточие, так же как после циклов и ветвлений. И так же как в циклах и ветвлениях тело класса пишется с табуляцией. Сейчас тут пусто, о чем говорит ключевое слово **pass**, но давайте это исправим. Первый, метод, который прямо точно необходимо написать в классе - это **метод инициализации**. Метод инициализации **init** (ещё его называют “**конструктором класса**”) описывает создание нового объекта этого класса. То есть все то, что будет происходит в момент **инициализации** объекта, когда мы будем создавать прямоугольник. В нашем случае **инициализация** означает создание конкретного прямоугольника с конкретными длиной и шириной. То есть мы будем говорить “хочу прямоугольник с длиной такой-то и шириной такой-то”, а Python вернет нам такой прямоугольник.

```
1 class Rectangle:
2     def __init__(self, width, length):
3         self.width = width
4         self.length = length
```

Чтобы реализовать это в Python, необходимо написать метод с названием **__init__**, который принимает self в качестве первого объекта и все что мы захотим в качестве остальных. Мы хотим длину и ширину, поэтому запишем их. Далее мы хотим, чтобы этим длина и ширина стали атрибутами класса. Как вы уже наверное поняли, обращение к классу изнутри (то есть “к самому себе”) реализуется через ключевое слово **self**. Далее мы можем получить любой атрибут класса передав его название через точку после **self**. Таким образом реализована “вложенность” принадлежности, про которую мы уже говорили ранее, в самом начале лекции. Количество уровней такой вложенности может быть столько сколько вам нужно. А в нашем случае, на инициализации, мы хотим просто-напросто хотим завести

атрибуты прямоугольника width и length. Теперь давайте создадим прямоугольник.

```
1 class Rectangle:
2     def __init__(self, width, length):
3         self.width = width
4         self.length = length
5 
6 rect = Rectangle(4, 5)
7 print(rect)
```

Прекрасно! Теперь, после исполнения строки 6, у нас есть конкретный прямоугольник, а именно: абстрактный объект, который описан двумя числами. Давайте получим его длину и ширину.

```
1 class Rectangle:
2     def __init__(self, width, length):
3         self.width = width
4         self.length = length
5 
6 rect = Rectangle(4, 5)
7 print(rect.width, rect.length)
```

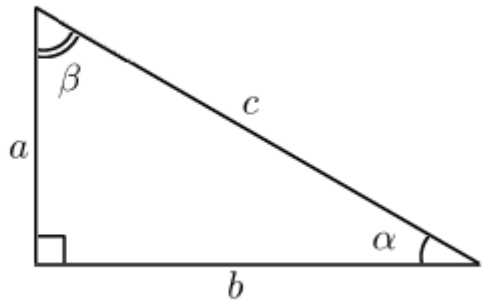
Великолепно! Теперь, когда мы знаем длины и ширину, будет несложно написать методы для вычисления площади, периметра и длины диагонали прямоугольника. Как мы помним, площадь прямоугольника вычисляется как произведение сторон, а периметр как сумма всех сторон. Также, напомню, что длину диагонали прямоугольника можно найти по теореме Пифагора, так как каждый диагональ в то же время является гипотенузой двух прямоугольных треугольников получившихся после её проведения внутри прямоугольника.

```
1  class Rectangle:
2      def __init__(self, width, length):
3          self.width = width
4          self.length = length
5  
6      def calc_area(self):
7          self.area = self.width * self.length
8          return self.area
9  
10     def calc_perimeter(self):
11         self.perimeter = 2 * (self.width + self.length)
12         return self.perimeter
13 
14     def calc_diag_len(self):
15         self.diag = (self.width ** 2 + self.length ** 2) ** 0.5
16         return self.diag
```

Методы *calc_area*, *calc_perimeter* и *calc_diag_len* вычисляют соответствующие величины, сохраняют их в соответствующий атрибут и возвращают их. Отмечу, что возврат значения через **return** в данном случае может быть избыточен. Важно решить для себя что именно данный метод должен исполнить: сохранение значения в атрибут, расчет и возврат значения или И ТО И ДРУГОЕ. В моем примере во всех трех методах я реализовал И ТО И ДРУГОЕ, для наглядной демонстрации, но мог и как не возвращать значения, а только сохранять его в атрибут, так и не сохранять значение в атрибут, но вместо этого возвращать его. Теперь попробуем вызвать данные методы:

```
1 class Rectangle:
2     def __init__(self, width, length):
3         self.width = width
4         self.length = length
5 
6     def calc_area(self):
7         self.area = self.width * self.length
8         return self.area
9 
10    def calc_perimeter(self):
11        self.perimeter = 2 * (self.width + self.length)
12        return self.perimeter
13 
14    def calc_diag_len(self):
15        self.diag = (self.width ** 2 + self.length ** 2) ** 0.5
16        return self.diag
17 
18 rect = Rectangle(4, 5)
19 print(f"Area: {rect.calc_area()}")
20 print(f"Perimeter: {rect.calc_perimeter()}")
21 print(f"Diagonal Length: {rect.calc_diag_len()}")
```

Давайте аккуратно прочитаем что происходит в конце данного скрипта. В 18 строке мы инициализируем прямоугольник со сторонами 4 и 5. В 19, 20 и 21 строках мы выводим на экран строки Area, Perimeter и Diagonal Length, плюс значения, которые возвращают вызовы методов *calc_area*, *calc_perimeter* и *calc_diag_len* одного и того же объекта *rect*. Попробуйте воспроизвести этот код на своем девайсе. Теперь мы напишем ещё один метод, после чего будем двигаться дальше. Последний метод будет рассчитывать углы между диагоналями прямоугольника. Вообще, для данного расчета существует больше одного подхода, но мы возьмем один, на основе косинусов, и рассмотрим его реализацию.



$$\sin\alpha = \frac{a}{c}$$

$$\cos\alpha = \frac{b}{c}$$

$$\mathrm{tg}\ \alpha = \frac{a}{b}$$

$$\mathrm{ctg}\ \alpha = \frac{b}{a}$$

Воспользуемся определением косинуса. **Косинус** угла в прямоугольном треугольника - это отношение прилежащего к этому углу катета к гипотенузе. Мы понимаем, что диагональ внутри прямоугольника делит его на два прямоугольных треугольника (то есть таких у которых один из углов равен 90 градусов). Также мы знаем, что зная две стороны в прямоугольном треугольнике, мы можем найти косинус угла между ними, а значит и сам угол. Для нас, это угол между диагональю и одной из сторон прямоугольника. Тогда мы можем провести вторую диагональ и вычислить искомый угол в маленьком треугольнике (одном из четырех), зная что сумма углов в треугольнике равна 180 градусов. А зная один из углов между диагоналями, легко найти и второй, зная что сумма этих углов равна 180 градусов. Давайте напишем код, реализующий данный алгоритм.

```python
class Rectangle:
    def __init__(self, width, length):
        self.width = width
        self.length = length

    def calc_area(self):
        self.area = self.width * self.length
        return self.area

    def calc_perimeter(self):
        self.perimeter = 2 * (self.width + self.length)
        return self.perimeter

    def calc_diag_len(self):
        self.diag = (self.width ** 2 + self.length ** 2) ** 0.5
        return self.diag

    def calc_diag_angles(self):
        """
        Calculates and returns two angles
        between the diagonals in degrees.
        """
        if not hasattr(self, 'diag'):
            self.calc_diag_len()

        cos_diag_length = self.length / self.diag
        angle_diag_length = math.acos(cos_diag_length)
        angle_diag_length = math.degrees(angle_diag_length)
        first_angle = 180 - (2 * angle_diag_length)
        second_angle = (360 - 2 * first_angle) / 2
        assert (second_angle * 2 + first_angle * 2) == 360
        return first_angle, second_angle
rect = Rectangle(3, 4)
rect.calc_diag_angles()
```

Во строках с 19-ой по 22-ую реализован т.н. **docstring** - документация метода. Как правило в **докстринге** описаны аргумента метода, возвращаемые значения, важные особенности и примеры использования. Это крайне важная и полезная информация, которая позволяет другим людям быстро понимать о чем ваш код. Разработчики как правило не очень любят рассказывать другим людям про то что они сделали в доступной форме. Но это КРИТИЧЕСКИ важно, если вы пишете код, который будет читать больше одного человека. Также, это становится важно лично для вас в случае когда методов достаточно много, и вы можете просто забыть что куда втыкалось и главное зачем. В строке 23, осуществляется проверка на наличия у данного объекта атрибута *diag*. Это нужно, поскольку для вычисления углов между диагоналями нужно знать длину диагоналей. Расчет длины у нас вынесен в отдельный метод. А это значит, что заранее из метода *calc_diag_angles* мы не знаем

была ли посчитана эта длина. Хорошо. А зачем проверка? Почему просто не пересчитывать её каждый раз? Пересчитывать её каждый раз не хочется, потому что зачем делать лишнее действие, если она уже была посчитана для этого прямоугольника однажды, и изменится она тоже не может. Также, такое пересчет может сыграть с вами плохую шутку, в случае, если расчет более сложный. Тогда скорость работы метода *calc_diag_angles* может значительно упасть. Поэтому, мы используем встроенный метод **hasattr**, который принимает объект и имя атрибута и возвращает True, если переданный атрибут существует в данном объекте

🔥 Важно! Метод hasattr ничего не знает про значения атрибута, но повествует исключительно о его наличии в объекте.

В строках с 26 по 30 производится собственно сам расчет. В 31 строке осуществляется проверка на правильность работы метода. Поскольку мы знаем что удвоенная сумма углов между диагоналями должна равняться 360 градусам, то если это не так, значит углы посчитаны неверно. Собственно, ключевое слово assert используется для проверки какого-то условия. Если условие вернет True, то ничего не произойдёт, а если False, то поднимется Exception типа AssertionError. Далее в 32 строке возвращаются два значения: первый и второй углы.

Теперь пройдемся по данному примеру в разрезе концепций.

1. При исполнении метода *calc_diag_angles* у нас нету доступа извне к промежуточным переменным таким как *cos_diag_length*, и *angle_diag_length*. Извне эти переменные нам просто не нужны. А также, мы не знаем какие assert-ы производятся во время расчетов, не говоря уже про содержание самих расчётов. В нашем примере это реализация идей **инкапсуляции**.
2. Прямоугольник описывается исключительно двумя сторонами. Он не является набором пикселей определенного цвета на определенном фоне. Он также не имеет координат ни двухмерном, ни в трехмерном, ни в каком-либо другом пространстве (поэтому, если мы например, захотим посчитать расстояния от него до какого-нибудь объекта, то просто мы не сможем этого сделать). Но для наших целей двух чисел вполне достаточно. Как вы уже догадались, именно так в нашем примере реализована **абстракция**.
3. Сейчас в методе init входные данные никак не обрабатываются и не проверяются. А что если бы мы захотели передавать ширину и длину в строках? Например словами “twelve” и “five” или числами в виде строковых переменных вроде “12” и “5”. Мы бы легко могли написать несколько строк, обрабатывающих такие случаи. Например в случае слов, было бы несложно найти или создать словарь который бы переводил слова в числа. А в случае чисел в строковом типе - это совсем просто. Нужно только конвертировать

string в int. Так можно было бы реализовать нашу любимую концепцию: **полиморфизм**.

4. И наконец, **наследование** в нашем случае явным образом не реализовано. На самом деле в Python все классы по умолчанию наследуют классу **object** и имеют тип **type**, если вы не укажете обратное. Поэтому говорят, что “в Python все является объектом”, так как у всего есть родительский класс. А родитель всех классов - это метакласс **type**. Подробнее об этом можно почитать в документации Python в разделе “Data Model”. В нашем случае - Rectangle тоже наследует метаклассу **type** (это легко проверить с помощью одноименного встроенного метода **type()**), поэтому **наследование** реализовано как бы неявно. Но если бы мы захотели создать какой-нибудь дочерний объект, например, квадрат, то было бы удобно унаследовать прямоугольник и кое-что переопределить. Тогда **наследование** было бы явным, таким каким нам нужно.

Прямоугольник - продолжение

Давайте теперь ещё больше усложним наш пример. Предположим, мы захотели использовать встроенный в Python оператор равенства (==) для того, чтобы сравнивать два прямоугольника по площади, а оператор суммы (+) для того, чтобы получать сумму площадей двух прямоугольников. Иначе говоря, что если мы хотим применить всю мощь идей полиморфизма для наших кастомных объектов? Проще простого! В Python мы можем использовать метод “**переопределения**” для того, чтобы **переопределить** операции для наших объектов. Но ДО того как что-либо менять, давайте проверим как эти операторы будут работать по дефолту:

```
rect_1 = Rectangle(2, 10)
rect_2 = Rectangle(4, 5)

rect_1.calc_area()
rect_2.calc_area()

rect_1 == rect_2
```

Попробуйте ответить, как вы думаете, что вернёт такой код для равенства?

```
rect_1 + rect_2
```

И вот такой для суммы?

Про равенство ответ простой: оно определено по умолчанию для всех объектов и проверяет совпадение адреса в памяти или, говоря простыми словами, проверяет

является ли этот объект слева тем же самым объектом что и объект справа. В терминах программирования, Python исполняет проверку id(object_1) == id(object_2), где id() - это встроенный метод возвращающий адрес объекта в памяти.

Про сумму ответ ещё проще: Python поднимет исключение типа TypeError, в который Python прямым текстом говорит нам, что не понимает как применить оператор суммы для нашего типа объекта Rectangle.

```
TypeError: unsupported operand type(s) for +: 'Rectangle' and 'Rectangle'
```

Теперь, когда мы знаем как эти операторы ведут себя по умолчанию, давайте переопределим их для нашего случая. Для того чтобы переопределить оператор равенства, необходимо прописать в нашем классе метод __eq__, а для суммы __add__. Вот код класса, который получится, если мы хотим сравнивать и суммировать площади. Я выкинул отсюда все вычисления кроме площади, так как сейчас они нам не нужны:

```python
class Rectangle:
    def __init__(self, width, length):
        self.width = width
        self.length = length

    def calc_area(self):
        self.area = self.width * self.length
        return self.area

    def __eq__(self, other):
        return self.area == other.area

    def __add__(self, other):
        return self.area + other.area
```

Методы __eq__ и __add__ принимают два аргумента: текущий объект (self) и другой объект (other), что логично, так как и операторы как сложения, так и равенства, принимают ровно два аргумента. В теле методов мы прописываем что именно они должны возвращать, в данном случае мы обращаемся к площади и в случае равенства сравниваем их и возвращаем результат сравнения, а в случае сложения - складываем и возвращаем сумму. Попробуйте сами проверить что теперь вернет вот этот код, который уже был написан выше.

```
rect_1 = Rectangle(2, 10)
rect_2 = Rectangle(4, 5)

rect_1.calc_area()
rect_2.calc_area()

rect_1 == rect_2, rect_1 + rect_2
```

А теперь попробуйте исполнить этот же код с другими прямоугольниками и проверить результат.

Структурно-процедурная реализация

А что если бы мы не использовали ООП и оставались бы в рамках структурно-процедурной реализации? Тогда, мы бы не могли описать объект “Rectangle”, но нам все ещё было бы необходимо разработать процедуры для вычисления площади, периметра, длины диагонали и углов между длинами. Давайте рассмотрим пример реализации в структурной и процедурной парадигмах.

```python
import math

def calc_area(width, length):
    return width * length

def calc_perimeter(width, length):
    return 2 * (width + length)

def calc_diag_len(width, length):
    return (width ** 2 + length ** 2) ** 0.5

def calc_diag_angles(length, diag):
    cos_diag_length = length / diag
    angle_diag_length = math.acos(cos_diag_length)
    angle_diag_length = math.degrees(angle_diag_length)
    first_angle = 180 - (2 * angle_diag_length)
    second_angle = (360 - 2 * first_angle) / 2
    assert (second_angle + first_angle) == 180
    return first_angle, second_angle

rect = (20, 30) ## инициализация прямоугольника
area = calc_area(*rect)
perimeter = calc_perimeter(*rect)
diag = calc_diag_len(*rect)
first_angle, second_angle = calc_diag_angles(rect[0], diag)

print(area, perimeter, diag, first_angle, second_angle)
```

Сходу видим в данном примере, что в сигнатурах функций дублируются одни и те же переменные. Но самое тут не это, а то неудобство с которым мы столкнемся в случае если прямоугольников не один, а несколько. В этом случае код выглядел бы как-то так:

```
rect_1 = (20, 30) ## инициализация прямоугольника 1
rect_2 = (4, 5)   ## инициализация прямоугольника 2
rect_3 = (5, 12)  ## инициализация прямоугольника 3

area_1 = calc_area(*rect_1) ## площадь прямоугольника 1
area_2 = calc_area(*rect_2) ## площадь прямоугольника 2
area_3 = calc_area(*rect_3) ## площадь прямоугольника 3

perimeter_1 = calc_perimeter(*rect_1) ## периметр прямоугольника 1
perimeter_2 = calc_perimeter(*rect_2) ## периметр прямоугольника 2
perimeter_3 = calc_perimeter(*rect_3) ## периметр прямоугольника 3

diag_1 = calc_diag_len(*rect_1) ## диагональ прямоугольника 1
diag_2 = calc_diag_len(*rect_2) ## диагональ прямоугольника 2
diag_3 = calc_diag_len(*rect_3) ## диагональ прямоугольника 3

first_angle_1, second_angle_1 = \
    calc_diag_angles(rect_1[0], diag_1) ## углы между диагоналями прямоугольника 1
first_angle_2, second_angle_2 = \
    calc_diag_angles(rect_2[0], diag_2) ## углы между диагоналями прямоугольника 2
first_angle_3, second_angle_3 = \
    calc_diag_angles(rect_3[0], diag_3) ## углы между диагоналями прямоугольника 3
```

Тут же возникает неприятное это чувство как-будто все перемешалось в кашу. И конечно же очевидно множественное дублирование кода. Поэтому следующий вопрос который возникает - это вопрос хранения данных (в нашем случае - длин, ширин и вычисленных значений). Как лучше сделать? Прямоугольников теперь много, может будем хранить их в массиве? Например вот так:

```
rectangles = [(20, 30), (4, 5), (5, 12)]
```

Но как тогда к обращаться к их сторонам - по индексу в этом массиве? То есть получается каждый прямоугольник имеет индекс. А это точно удобно? А может лучше их хранить в словаре - тогда у каждой пары длины и ширины хотя бы будет имя. Например вот так:

```
rectangles = {
    'rect_1': (20, 30),
    'rect_2': (4, 5),
    'rect_3': (5, 12),
}
rectangles['rect_1']
```

Можно и так конечно. А как убедиться что то, что мы получили из словаря - это именно пара длины и ширины конкретного прямоугольника, а не просто какие-то числа которые оказались у нас в руках случайно? И как тогда инициализировать объект - просто заводить пару ключ и значение в словаре? И ещё, как понять что из пару есть длина, а что ширина? И как не запутаться в числах, если атрибутов вдруг будет становиться больше, например, когда мы вычислим диагонали и углы? Тогда нужно будет создавать словарь словарей?

```
rectangles = {
    'rect_1': {'width': 20,
               'length': 30},
    'rect_2': {'width': 4,
               'length': 5},
    'rect_3': {'width': 5,
               'length': 12},
}
print("Ширина первого прямоугольника:", \
      rectangles['rect_1']['width'])
print("Длина второго прямоугольника:" , \
      rectangles['rect_2']['length'])
```

То есть уже хочется как-то эти числа сгруппировать, но чувствуется некое нагромождение. Ирония в том, что под капотом у ООП Python конечно же реализованы древовидные структуры вроде таких же словарей, но использовать их как объекты и атрибуты - заметно удобнее и приятнее с точки зрения синтаксиса. Вот так:

```
rect_1 = Rectangle(20, 30)
rect_2 = Rectangle(4, 5)

print("Ширина первого прямоугольника:", \
      rect_1.width)
print("Длина второго прямоугольника:" , \
      rect_2.length)
```

Очень важно также добавить, что код становится понятнее для сторонних читателей.

Таким образом, структурным подходом можно обойтись, если логика остается простой, последовательность действий - линейной. И самое главное - объектов не становится больше одного. Иначе их необходимо как-то хранить и скорее всего проще использовать ООП. Таким образом, ООП становится верным союзником, в случаях, когда:

1. Последовательность действий нелинейна
2. Логика сложная
3. Объектов много: тогда их методы и атрибуты удобно хранить внутри этих объектов

В каждом из этих случаев, ООП действительно может упростить жизнь и лично вам и тем, кто будет взаимодействовать с вашим кодом в дальнейшем.

Преимущества и недостатки ООП

Резюмируя, рассмотрим более детально преимущества ООП подхода в разработке:

- Экономия времени на разработке:
    - Классы проще и удобнее читать (а ещё это красиво!)
    - Наследование позволяет переиспользовать код, создавая родительские классы
- Безопасность за счёт инкапсуляции: атрибуты принадлежат своим объектам и могут быть недоступны для вызова извне своего объекта
- Наглядность и упрощение с помощью абстракции

К недостаткам ООП можно отнести:

- Высокий порог входа для программиста: считается что чтобы понять и научиться ООП нужно больше времени, так как программисту необходимо научиться мыслить “объектно”. Но на самом деле это не является универсальным правилом и порог входа в ООП лично для вас может сильно отличаться от других людей. Поэтому проще ориентироваться на собственный вкус и текущих условий задачи.
- Скорость исполнения: ООП код обычно работает немного медленнее процедурного

Итоги лекции

В сегодняшней лекции мы разобрали объектно-ориентированное программирование.

- Погрузились в контекст вокруг ООП

- Выяснили что такое программа в ООП, что такое объект, класс, метод, атрибут и как это работает
- Свойства объектов и атрибутов
- Концепции ООП
- История ООП вкратце
- Примеры ООП кода
- Преимущества и недостатки парадигмы

Если у вас остались вопросы, вы можете написать их мне в Telegram: @alexlevinML

Спасибо что прослушали это лекцию и увидимся в следующей!

—

Что можно почитать еще?

1. Типы в языках программирования - Пирс Бенджамин
2. Dahl, Nygaard - The Birth of Simula
3. IBM System 360/370 Compiler and Historical Documentation
4. The Early History Of Smalltalk
5. Bjarne Stroustrup - A History of C++

Используемая литература

1. Is Go an Object Oriented Language
2. Статья про Полиморфизм на Medium
3. Официальная документация Python, раздел “Data Model”

—