ООП, Класи, Об'єкти, Інтерфейси

Структура

Що таке ООП?

Раніше ми розглядали різні сутності, такі як: `Int`, `Boolean`, `String` та ін. Прийшло час розібратися, що таке типи (або об'єкти у більш широкому розумінні цього терміну). Наприклад, щоб розуміти, як працюють числа з плаваючою точкою (Числа з комою, якщо простіше) з цілими числами (наш Int, від слова Integer). Як вони
порівнюються між собою і те, що за супертип у них такий Number? Стоп, а що таке супертип?

Давайте по порядку:
Як я вже казав Kotlin – це об'єктно-орієнтована мова програмування.
Що означає, що все, що ви робитимете буде будуватися на об'єктах.

Об'єкт

Щоб більше розуміти як працюють об'єкти у програмуванні, давайте введемо приклад з реального життя: наприклад, у вас є домашній вихованець. Нехай це буде кіт.
Розглянемо його як об'єкт. У нього є назва (тобто його ідентифікатор серед інших вихованців), є якісь властивості (наприклад, його ім'я, вік та вага), є функції (наприклад, нявкання або засинання), є події (наприклад, знову ж таки, коли кіт нявкнув або заснув).

Давайте візуалізуємо нашого кота:
![Кіт][images/oop_cat_ua.excalidraw.svg]

Властивості

Властивість, грубо кажучи, це те саме, що й змінна, тільки вона прив'язана до об'єкту.

Функції

Раніше ми розглядали, що таке функції. Єдине, що відрізняється у даному випадку - це те, що нам потрібен об'єкт (його екземпляр) щоб її викликати.

Екземпляр? Так як об'єкт - це самостійна структура, об'єкт може існувати в декількох варіантах (наприклад, у вас може бути більше одного кота). Це називають екземплярами об'єкта (тобто кількома варіаціями об'єкта).

Події

Припустимо, що нашому коту потрібен спокійний і тривалий сон.
І нам потрібно поводитися тихіше, коли наш кіт у режимі сну.
Для цього ми повинні встановити візуальний контакт та дочекатися моменту, коли наш кіт засне.
У програмуванні термін спостереження називають «слухачем». Слухач отримує деяку інформацію, яку у ООП називають «повідомленням».

Слухач - сутність (aka об'єкт) з однією або декількома функціями, що описують наші «повідомлення».

Подія - це повідомлення з якогось фрагмента коду іншому фрагменту коду.

Інкапсуляція

Як ми вже розглянули, об'єкти мають властивості та функції. Що ж нам робити, коли потрібно змінити якісь властивості всередині об'єкта? Наприклад, уявіть, що ми ведемо інстаграм-профіль нашого кота і нам потрібно змінити там його вік або щось інше. На виручку, до нас приходить інкапсуляція.
Що таке інкапсуляція? Інкапсуляція - це властивість об'єкта змінювати свій вміст (тобто свої внутрішні характеристики).
Повертаючись до кота, щоб змінити його вік, ми умовно створюємо функцію оновити вік, що змінює властивість зсередини.

Спадкування (або що там за абстракціями)

Ускладнимо завдання: у нас є притулок з домашніми тваринами, і нам потрібно надати уніфіковану інформацію про кожну тварину.
Якщо ви раніше створювали таблиці в тому ж Excel, ви, швидше за все, розумієте, як ви можете виділити параметри кожного вихованця.

Що ж таке спадкування? **Спадкування** - це властивість об'єкта набувати риси *іншого об'єкта*.

Як же вирішуватимемо завдання? Давайте спочатку виділимо, які вихованці у нас взагалі є:

- собаки
- коти
- папуги

Тепер нашим завданням є знайти загальні властивості даних об'єктів (вихованців) щоб виділити їх у більш загальну сутність.
Перше, що швидше за все, спало на думку — це імена. У всіх домашніх тварин є якісь імена. Відразу ж після цього, буде неважко додумати деякі інші властивості: скільки їм років, їх опис (може, наприклад, це що говорящий папуга або кіт) та інші їх атрибути (властивості).
![Наслідування][images/oop_inheritance1_ua.excalidraw.svg]
Вихованець тут — загальна сутність.

Тепер, коли ми маємо загальну абстрактну сутність (aka об'єкт), ми можемо підігнати наших котів, папуг і собак під цю сутність.
Спадкування гарантує, що виділена абстракція (aka загальна сутність) буде задана об'єктом, що її успадковує. З цього випливає, що якщо ми розмістимо список наших вихованців на нашому сайті, ми не отримаємо того, що наприклад у папуг немає зазначеного віку та ін.

Абстрагування

Беремо олімп щодо ускладнення наших завдань: нам потрібно надати ветеринарний паспорт заінтересованим особам. Але, паспорти, у нас, умовно, перебувають у відмінному від знаходження вихованців місці. В іншому відділі, наприклад.
Не всім потрібно бачити паспорт доти, доки вони не оберуть вихованця, наприклад, за їхньою породою.
З'являється завдання абстрагувати не потрібні, спочатку, властивості. Для цього Існує поняття абстрагування.
Абстрагування - уявний процес відділення властивостей об'єктів до якогось ідеального стану зручного для нас.
У нашому випадку ми відокремили зайву інформацію для першообивателя.

Поліморфізм

Вкотре ускладнимо завдання з нашими котиками та папугами: раніше було сказано, ми не тримаємо вет. паспорти поруч із папугою, а десь в іншому відділі. І коли потребується паспорт, ми дістанемо його з нашого 'сховища' (іншого відділу). Для розв'язання цього завдання є різні способи, що об'єднані одним терміном – поліморфізм.

Давайте введемо новий умовний об'єкт «Сховище», де у нас зберігатимуться паспорти наших вихованців. Додамо туди функцію «отримати Паспорт» з аргументом (aka параметром) «Кіт» (наш об'єкт, де зберігається інформація про нашого коте). Ну і
продублюємо ці функції так само для «Папуга» та «Собака» (так само наші об'єкти).

Що ж це виходить? Є три функції з однаковим ім'ям (aka ідентифікатором), що, за ідеєю, має бути заборонено мовою програмування (Aka компілятором). Але не так. Поліморфізм вирішує цю проблему, вводячи подібну можливість (насправді трохи складніше, але поки що зупинимося у цьому).

У програмуванні це називають *перевантаженням функції*. Коли функції мають однакові імена, але різний набір параметрів.

Реалізація ООП в Котлін

Після того, як ми обговорили теорію ООП, давайте перейдемо безпосередньо до Котліну.

Уявімо, що нам потрібно зберегти чи структурувати наші дані про наших улюблених котів чи кішок.
Нагадаю, що там у нашого кота є:
![Кіт][images/oop_cat_ua.excalidraw.svg]
Для опису використаємо `object`, про який зараз поговоримо.

Object

Одним із видів об'єктів у котлині є `object` (ось така ось тавтологія).
Чим відрізняється теоретичне поняття об'єкта від object?
Правильною відповіддю буде: нічим. Об'єкт є найбільш, мабуть, із простих варіантів об'єкту в Kotlin.

Чому? Він має лише одну варіацію (екземпляр) своєї структури.
Що ж, почнемо створювати нашого кота:

```
object Cat
```

Як можна зазначити, такий тип об'єктів ми записуємо за допомогою ключового слова `object`, після якого йде ім'я нашого об'єкту.

Властивості

Нумо додамо властивості (змінні) нашому об'єктові:

```
object Cat {
        val name: String = "Мася"
        val yearsOld: Int = 15 // людських, звичайно
        val weight: Double = 4.5
}
```

Властивості - це змінні на рівні об'єкта. Тобто змінні, що оголошено у його тілі.
Що ж, ми описали нашу улюблену Масю і тепер може брати з об'єкту інформацію про неї:

```
fun main() {
        val name = Cat.name
        println(name + " наш улюблений кіт!")
}
```

Воу-воу.. строка + строка?
Так, таке у програмуванні, називають конкатизацією строк (від англ. concat - об'єднувати). Логічно буде думати, що вона складає дві строки в одну з урахуванням їхнього вмісту.

Функції

Далі, якщо дивитися на структуру нашого кота, йдуть функції.
Оголошення функцій у класі / поза класом не відрізняється. Все по тій самій формулі:

```
object Cat {
        val name: String = "Мася"
        val yearsOld: Int = 15 // людських, звичайно
        val weight: Double = 4.5
        fun meow() {
                println(name + " тільки що нявкнув.")
        }
        fun sleep() {
                println(name + " тільки що задрімав.")
        }
}
```

Зробимо просте друкування того, що котик нявкнув/заснув. Все дуже просто!

Але, що робити, якщо у нас декілька котів? Об'єкт тут вже не варіант.
На допомогу нам приходить, один з видів об'єкта — **Клас**.
Клас може створюватися *в кількох екземплярах*, які будуть незалежними один від одного.

Клас

Давайте ж розглянемо класи лише на рівні мови.
Для початку створимо клас з тією самою назвою:

```
class Cat {
        val name: String
        val yearsOld: Int
        val weight: Double
        fun meow() {
                println(name + " тільки що нявкнув.")
        }
        fun sleep() {
                println(name + " тільки що задрімав.")
        }
}
```

Оголошення нічим не відрізняється від того, як ми оголошували змінні в тілі об'єкта або функції.
Але ось проблема! Структура нічого не знає про наших котів!
Ми, звичайно, можемо написати наступне (як це було раніше):

```
class Cat {
        val name: String = "Мася"
        val yearsOld: Int = 15
        val weight: Double = 4.5
        fun meow() {
                println(name + " тільки що нявкнув.")
        }
        fun sleep() {
                println(name + " тільки що задрімав.")
        }
}
```

Але, у нас кілька котів, як бути?
Завдання вирішується дуже легко логічно - нам потрібно зробити властивості довільними. Що потрібно для цього зробити? Створити конструктор!

Конструктор

Якщо ви знаєте англійську, ви швидше за все, зрозуміли, що конструктор (англ. constructor) щось конструює (збирає, створює та інші варіанти перекладу). Використовується для того ж, для чого і аргументи у функцій (насправді, конструктор - це така ж функція, тільки трохи покращена для своїх задач) — для передачі якоїсь необхідної інформації нашому коду (структурі) (наприклад, структура спочатку не знає, як звати нашого кота, ми повинні задати йому ім'я).
За такою ж формулою, як і з функцією, задаємо параметри класу:

```
class Cat(name: String, yearsOld: Int, weight: Double) {...}
```

Параметри класу видно лише всередині класу, але не в функціях (пізніше ми поговоримо про області видимості).
Давайте задамо нашим властивостям довільні значення з конструктора:

```
class Cat(name: String, yearsOld: Int, weight: Double) {
        val name: String = name
        val yearsOld: Int = yearsOld
        val weight: Double = weight
}
```

Варто зазначити, що подібне повторення назв допускається, оскільки конфлікти імен неможливі через те, що параметри видно лише всередині тіла класу.
Подібний код явно надто заскладний для такого тривіального завдання! Нам ліньки стільки писати, як бути?

Котлін вирішує цю проблему за рахунок того, що допускає оголошення властивостей в конструкторах!

```
class Cat(val name: String, val yearsOld: Int, val weight: Double)
```

Вже куди простіше, чи не так?
Варто зазначити, що властивості видно будь-якому місцю, що відноситься безпосередньо до структури (наприклад, властивості будуть видні при звертанні до екземпляру об'єкта).

Щоб використати даний об'єкт, ми викликаємо його конструктор як будь-яку іншу функцію:

```
fun main() {
        val cat = Cat("Кіттик", 20, 5.1)
        // ім'я доступне тут
        println(cat.name)
        // але, не доступно тут:
        println(Cat.name) // потрібен екземпляр класа, що створюється через
конструктор
}
```

Функції

Функції, знову ж, будуть ідентично записані, як і у звичайного об'єкта:

```
class Cat(val name: String, val yearsOld: Int, val weight: Double) {
        fun meow() {
                println(name + " тільки що нявкнув.")
        }
        fun sleep() {
                println(name + " тільки що задрімав.")
        }
}
```

Тепер же перейдемо до наших подій.
Нагадаю, що подія - це просто об'єкт, який має деякі функції, що описують наше "повідомлення".
Наприклад, щось таке:

```
object Event {
        fun whenMeow() {
        println("Він зробив няв!")
        }
        fun whenFellAsleep() {
                println("Поводимося тихо! Він спить.")
        }
}
```

І можемо це викликати з наших функцій, що знаходяться у Коті:

```
class Cat(val name: String, val yearsOld: Int, val weight: Double) {
        fun meow() {
                Event.whenMeow()
        }
        fun sleep() {
                Event.whenFellAsleep()
        }
}
```

Це і буде варіацією подій.
Але, як ви встигли подумати, такий варіант відтинає можливість мати довільні слухачі. Тобто якщо нам знадобиться для одного окремого кота поміняти "слухач" (наприклад повідомлення всередені) - ми не зможемо.
І на допомогу нам приходить інтерфейс!

Інтерфейс

Інтерфейс (англ. interface) — програмна структура, що визначає відношення між об'єктами, які поділяють певну поведінкову множину і не пов'язані ніяк інакше.

Раніше ми розглядали абстракції по відношенню до різних вихованців (це ми також розберемо), але поки давайте вирішимо більш насущну задачу.

По суті, інтерфейс не має ніякого стану (тобто не може інкапсулюватись: змінювати властивості та ін.). Цей тип «об'єктів» лиш описує те, як інші об'єкти будуть поводитись, тобто:

```
interface Event {
        fun whenMeow()
        fun whenFellAsleep()
}
```

Ми не задаємо якоїсь логіки, а лиш описуємо те, що об'єкт, який наслідує (імплементує) інтефейс буде робити:

```
object DefaultEvent : Event { // наслідування відбувається за допомогою
символа ':'
        // задавання методів, що були описані в інтерфейсі, відбувається за
допомогою ключового слова 'override'
        override fun whenMeow() {
                println("Він зробив няв!")
        }
        // задавання методів, що були описані в інтерфейсі, відбувається за
допомогою ключового слова 'override'
        override fun whenFellAsleep() {
                println("Поводимося тихо! Він спить.")
        }
}
```

Що ж щодо нашої задачі, все що нам потрібно тепер, це передати в клас інтерфейс, який можна буде підмінити різними варіаціями за допомогою наслідування.

```
class Cat(val name: String, val yearsOld: Int, val weight: Double, private
val event: Event) { // передали в конструктор інтерфейс
        fun meow() {
                event.whenMeow() // викликали функцію на екземплярі класа, що
наслідує Event
        }
        fun sleep() {
                event.whenFellAsleep() // викликали функцію на екземплярі
класа, що наслідує Event
        }
}
```

Тепер же, перейдімо до «вихованців», а якщо бути точнішим, до їх різновиду.

> TODO