Введение в контексно-ориентированное программирование на Kotlin

Это перевод статьи[An introduction to context-oriented programming in Kotlin]

В этой статье я постараюсь описать новое явление, которое возникло как побочный результат стремительного развития языка Kotlin. Речь идет о новом подходе к проектированию архитектуры приложений и библиотек, который я буду называть контекстно-ориентированным программированием.

Несколько слов о разрешении функций

Как хорошо известно, существует три основных парадигмы программирования (*примечание Педанта*: есть и другие парадигмы):

- Процедурное программирование
- Объектно-ориентированное программирование
- Функциональное программирование

Все эти подходы так или иначе работают с функциями. Давайте посмотрим на это с точки зрения разрешения функций, или диспетчеризации их вызовов (имеется в виду выбор функции, которая должна быть использована в данном месте). Для процедурного программирования характерно использование глобальных функций и их статическое разрешение, основанное на имени функции и типах аргументов. Конечно, типы могут быть использованы только в случае статически типизированных языков. Например, в Python функции вызываются по имени, и если аргументы неправильные, в конце концов возбуждается исключение в рантайме (во время выполнения программы). Разрешение функций в языках с процедурным подходом основано только на имени процедуры/функции и ее параметрах, и в большинстве случаев делается статически.

Объектно-ориентированный стиль программирования ограничивает области видимости функций. Функции не глобальны, вместо этого они являются частью классов, и могут быть вызваны только для экземпляра соответствующего класса (*примечание Педанта*: некоторые классические процедурные языки имеют модульную систему и, значит, области видимости; процедурный язык != C). Конечно, мы всегда можем заменить функцию-члена класса глобальной функцией с дополнительным аргументом, имеющим тип вызываемого объекта, но с синтаксической точки зрения разница довольно значительна. Например, в этом случае методы сгруппированы в классе, к которому они обращаются, и поэтому более ясно видно какое поведение обеспечивают объекты данного типа. Конечно, наиболее важны здесь инкапсуляция, благодаря которой некоторые поля класса или его поведение могут быть приватными и доступными только членам этого класса (вы не можете обеспечить этого в чисто процедурном подходе), и полиморфизм, благодаря которому фактически используемый метод определяется не только на основе имени метода, но и на основе типа объекта, из которого он вызывается. Диспетчеризация вызова метода в объектно-ориентированном подходе зависит от типа объекта, определяемого в рантайме, имени метода, и типа аргументов на этапе компиляции.

Функциональный подход не приносит чего-то принципиально нового в плане разрешения функций. Обычно функционально-ориентированные языки имеют лучшие правила для разграничения областей видимости (*примечание Педанта*: еще раз, C - это еще не все процедурные языки, есть такие, в которых области видимости хорошо разграничены), которые позволяют проводить более скрупулезный контроль видимости функций на основе системы модулей, но кроме этого, разрешение проводится во время компиляции основываясь на типе аргументов.

Что такое *this*?

В случае объектного подхода, при вызове метода у объекта,у нас есть его аргументы, но кроме того мы имеем явный (в случае Python) или неявный параметр, представляющий экземпляр вызываемого класса (здесь и далее все примеры написаны на Kotlin):

```
class A{
    fun doSomething(){
        println("    $this")
    }
}
```

Вложенные классы и замыкания все несколько усложняют:

```
interface B{
    fun doBSomething()
}

class A{
    fun doASomething(){
        val b = object: B{
            override fun doBSomething(){
                println("    $this  ${this@A}")
            }
        }
        b.doBSomething()
    }
}
```

В данном случае есть два неявных *this* для функции*doBSomething*- один соответствует экземпляру класса *B*, а другой возникает от замыкания экземпляра *A*. То же самое происходит в намного более часто встречающемся случае лямбда-замыкания. Важно отметить, что *this* в данном случае работает не только как неявный параметр, но и как область или контекст для всех функций и объектов, вызываемых в лексической области определения. Так что метод*doBSomething*на самом деле имеет доступ к любым, открытым или закрытым, членам класса*A*, так же как и к членам самого *B*.

А вот и Kotlin

Kotlin дает нам совершенно новую "игрушку" - функции-расширения.(*Примечание Педанта*: на самом деле не такие уж новые, в C# они тоже есть). Вы можете определить функцию вроде *A.doASomething()* где угодно в программе, не только внутри *A*. Внутри этой функции у нас есть неявный *this*-параметр, называемый получателем (receiver), указывающий на экземпляр *A* на котором метод вызывается:

```
class A

fun A.doASomthing(){
    println(" -   $this")
}

fun main(){
    val a = A()
    a.doASomthing()
}
```

У функций-расширений нет доступа к закрытым членам их получателя, так что инкапсуляция не нарушается.

Следующая важная вещь, которая есть в Kotlin - блоки кодас получателями. Можно запустить произвольный блок кода используя что-нибудь в качестве получателя:

```
class A{
    fun doInternalSomething(){}
}

fun A.doASomthing(){}

fun main(){
    val a = A()
    with(a){
        doInternalSomething()
        doASomthing()
    }
}
```

В этом примере обе функции можно было вызвать без дополнительного "*a.*" в начале, потому что функция *with* помещает весь код последующего блока внутрь контекста *a*. Это значит, что все функции в этом блоке вызываются так, как если бы они вызывались на (явно переданном) объекте *a*.

Окончательный на этот момент шаг к контекстно-ориентированному программированию - возможность объявлять расширения как члены класса.В этом случае функция-расширение определяется внутри другого класса, вот так:

```
class B

class A{
    fun B.doBSomething(){}
}

fun main(){
    val a = A()
    val b = B()
    with(a){
        b.doBSomething() //
    }
    b.doBSomething() //
}
```

Важно, что здесь*B* получает некоторое новое поведение, но только когда находится в конкретном лексическом контексте. Функция-расширение является обычным членом класса *A*. Это значит, что разрешение функции делается статически на основе контекста, в котором она вызывается, но настоящая реализация определяется экземпляром *A*, передаваемым в качестве контекста. Функция может даже взаимодействовать с состоянием объекта *a*.

Контекстно-ориентированная диспетчеризация

В начале статьи мы обсудили разные подходы кдиспетчеризации вызовов функций, и это было сделано не просто так. Дело в том, что функции-расширения в Kotlin позволяют работать с диспетчеризацией по-новому. Теперь решение о том, какая конкретно функция должна быть использована, основано не только на типе ее параметров, но и на лексическом контексте ее вызова. То есть то же самое выражение в разных контекстах может иметь разное значение. Конечно, с точки зрения реализации ничего не меняется, и у нас по-прежнему есть явный объект-получатель, который определяет диспетчеризацию для своих методов и расширений, описанных в теле самого класса (member extensions)- но с точки зрения синтаксиса, это другой подход.

Давайте рассмотрим, как контекстно-ориентированный подход отличается от классического объектно-ориентированного, на примере классической проблемы арифметических операций над числами в Java. Класс *Number* в Java и Kotlin является родительским для всех чисел, но в отличие от специализированных чисел вроде Double, он не определяет своих математических операций. Так что нельзя писать, например, так:

```
val n: Number = 1.0

n + 1.0 //  `plus`     `Number`
```

Причина здесь в том, что невозможно согласованно определить арифметические операции для всех числовых типов. К примеру, деление целых чисел отличается от деления чисел с плавающей точкой. В некоторых особых случаях пользователь знает, какой тип операций нужен, но обычно нет смысла определять такие вещи глобально. Объектно-ориентированным (и, на самом деле, функциональным) решением было бы определить новый тип-наследник класса*Number*, нужные операции в нем, и использовать его где необходимо (в Kotlin 1.3 можно использовать встраиваемые (inline) классы). Вместо этого, давайте определим контекст с этими операциями и применим его локально:

```
interface NumberOperations{
    operator fun Number.plus(other: Number) : Number
    operator fun Number.minus(other: Number) : Number
    operator fun Number.times(other: Number) : Number
    operator fun Number.div(other: Number) : Number
}

object DoubleOperations: NumberOperations{
    override fun Number.plus(other: Number) = this.toDouble() + other.toDouble()
    override fun Number.minus(other: Number) = this.toDouble() - other.toDouble()
    override fun Number.times(other: Number) = this.toDouble() * other.toDouble()
    override fun Number.div(other: Number) = this.toDouble() / other.toDouble()
}

fun main(){
    val n1: Number = 1.0
    val n2: Number = 2
    val res = with(DoubleOperations){
        (n1 + n2)/2
    }

    println(res)
}
```

В этом примере расчет *res* делается внутри контекста, который определяет дополнительные операции. Контекст не обязательно определять локально, вместо этого он может быть передан неявно как получатель функции. Например, можно сделать так:

```
fun NumberOperations.calculate(n1: Number, n2: Number) = (n1 + n2)/2

val res = DoubleOperations.calculate(n1, n2)
```

Это означает, что логика операций внутри контекста полностью отделена от реализации этого контекста, и может быть написана в другой части программы или даже в другом модуле. В этом простом примере контекст - это синглтон без состояния, но можно использовать и контексты с состоянием.

Также стоит помнить, что контексты могут быть вложенными:

```
with(a){
    with(b){
        doSomething()
    }
}
```

Это дает эффект комбинирования поведений обоих классов, однако данную фичу на сегодняшний день трудно контролировать из-за отсутствия расширений с множественными получателями (KT-10468).

Мощь явных корутин (coroutines)

Один из лучших примеров контекстно-ориентированного подхода использован в библиотеке Kotlinx-coroutines. Объяснение идеи можно найти в статье Романа Елизарова. Здесь я только хочу подчеркнуть, что*CoroutineScope* - это случай контекстно-ориентированного дизайна с контекстом, имеющим состояние.*CoroutineScope*играет две роли:

- Он содержит*CoroutineContext*, который нужен для запуска корутин и наследуется когда запускается новая сопрограмма.
- Он содержит состояние родительской корутины, позволяющее отменить ее в случае, если порожденная сопрограмма выкидывает ошибку.

Также, структурированная конкурентность (structured concurrency)предоставляет отличный примерконтекстно-ориентированной архитектуры:

```
suspend fun CoroutineScope.doSomeWork(){}

GlobalScope.launch{
    launch{
        delay(100)
        doSomeWork()
    }
}
```

Здесь*doSomeWork* - это контекстная функция, но определенная за пределами ее контекста. Методы *launch* создают два вложенных контекста, которые эквивалентны лексическим областям соответствующих функций (в данном случае оба контекста имеют один и тот же тип, поэтому внутренний контекст затеняет внешний). Хорошей отправной точкой для изучения сопрограмм в Kotlin является официальное руководство.

DSL

Существует широкий класс задач для Kotlin, на которые обычно ссылаются как на задачи построения DSL (Domain Specific Language). Под DSL при этом понимается некоторый код, обеспечивающий дружественный пользователю построитель (builder) какой-то сложной внутри структуры. На самом деле использование термина DSL здесь не совсем корректно, т.к. в таких случаях просто используется базовый синтаксис Kotlin без каких-либо специальных ухищрений - но давайте все-таки использовать этот распространенный термин.

DSL-построители в большинстве случаев контекстно ориентированы. Например, если вы хотите создать HTML-элемент, надо в первую очередь проверить, можно ли добавлять этот конкретный элемент в данное место. Библиотекаkotlinx.html|достигает этого предоставлением основанных на контексте расширений классов, представляющих определенный тег. По сути, вся библиотека состоит из контекстных расширений для существующих элементов DOM.

Другой пример - построитель GUI TornadoFX. Весь построитель графа сценыустроен как последовательность вложенных контекст-построителей, где внутренние блоки отвечают за построение детей для внешних блоковили подстройку параметров родителей. Вот пример из официальной документации:

```
override val root = gridPane{
    tabpane {
        gridpaneConstraints {
            vhGrow = Priority.ALWAYS
        }
        tab("Report", HBox()) {
            label("Report goes here")
        }
        tab("Data", GridPane()) {
            tableview<Person> {
                items = persons
                column("ID", Person::idProperty)
                column("Name", Person::nameProperty)
                column("Birthday", Person::birthdayProperty)
                column("Age", Person::ageProperty).cellFormat {
                    if (it < 18) {
                        style = "-fx-background-color:#8b0000; -fx-text-fill:white"
                        text = it.toString()
                    } else {
                        text = it.toString()
                    }
                }
            }
        }
    }
}
```

В этом примере лексическая область определяет свой контекст (что логично, т.к. он представляет раздел GUI и его внутреннее устройство), и имеет доступ к родительским контекстам.

Что дальше: множественные получатели

Контекстно-ориентированное программирование дает разработчикам Kotlin множество инструментов и открывает новый способ дизайна архитектуры приложений. Нужно ли нам что-то еще?

Вероятно, да.

На данный момент разработка в контекстном подходе ограничена тем фактом, что нужно определять расширения, чтобы получить какое-то ограниченное контекстом поведение класса. Это нормально, когда речь идет о пользовательском классе, но что если мы хотим то же самое для класса из библиотеки? Или если мы хотим создать расширение для уже ограниченного в области поведения (например, добавить какое-то расширение внутрь *CoroutineScope*)? На данный момент Kotlin не позволяет функциям-расширениям иметь более одного получателя. Но множественные получатели можно было бы добавить в язык, не нарушая обратной совместимости. Возможность использованиямножественных получателей сейчас обсуждается(KT-10468) и будет оформлена в виде KEEP-запроса (UPD: уже оформлена). Проблема (или, может быть, фишка) вложенных контекстов - в том, что они позволяют покрыть большинство, если не все, варианты использования классов типов (type-classes), другой очень желанной из предложенных фич. Довольно маловероятно, что обе эти фичи будут реализованы в языке одновременно.

Дополнение

Я хочу поблагодарить нашего штатного Педанта и любителя HaskellАлексея Худяковаза его замечания по тексту статьи и поправки по моему достаточно вольному использованию терминов. Также благодарю Илью Ръженкова за ценные замечания и вычитку английской версии статьи.