An introduction to context-oriented programming in Kotlin

In this article I will try to describe a new interesting phenomenon which appeared as a by-product of fascinating progress made by Kotlin development team. Namely, the new approach to library and application architecture design, which I call context-oriented programming.

A few words about function resolution

It is well known, that there are three main programming paradigms (*pedant's comment: there are other paradigms as well*):

- Procedural programming
- Object-oriented programming
- Functional programming

All those approaches work with functions with one or another way. Let's look on them from the point of function resolution or dispatch (meaning which function will be used in which case). Procedural programming tends to use global functions and resolve them statically based on the name and argument type. Of course types have meaning only in case of statically typed languages. In python, for example one calls function by name and if the parameters are wrong, an exception will be thrown in runtime. The function resolution in procedural language is based on procedure/function name and its parameters only and in most cases static.

Object-oriented programmingstyletends to limit function visibility. Functions are not global, but instead are members of objects which means that they could be called only on the instance of a specific object (*pedant's comment: some classic procedural languages have module system and therefore visibility scopes, procedural language!= C*). Of course one can always replace a object member function with global function with additional argument with the type of the calling object, but from syntactic point of view the difference is rather large. For example, now methods are grouped by the object they are called upon and so one could clearly see what methods or behaviors are supported by this type of object. Of course, the most important part is the encapsulation which means that some of object fields or behaviors could be private and accessible only from this object members (you can't do it in purely procedural approach) and polymorphism, which means that actually used method is decided not based upon the method name, but also on runtime type of the object it is called upon. Object-oriented dispatch is based on caller runtime type, method name and compile-time types of arguments.

Functional programming does not bring anything principally new in terms of function resolution. Usually, functional-oriented languages have better scoping rules (*peda nt's comment: again, not all procedural languages are C, so there are some with good scoping*), which allow to perform more fine-grained visibility control for functions based on module system, but overwise, the dispatch is performed based on compile-time types of arguments.

What isthis?

In case of object programming, when we call an object method, we have all of its parameters, but additionally we have an explicit (in case of Python) or implicit parameter representing the instance of calling class (here and later all examples are written in Kotlin):

```
class A{
    fun doSomething(){
        println("This method is called on $this")
    }
}
```

Nested classes and closures complicate things a bit:

```
interface B{
    fun doBSomething()
}

class A{
    fun doASomething(){
        val b = object: B{
            override fun doBSomething(){
                println("This method is called on $this inside ${this@A}")
            }
        }
        b.doBSomething()
    }
}
```

In this case there are two implicitthisparameters for function`doBSomething`. One comes form the instance of class`B`and another one comes form enclosing`A`instanc e. The same works for much more common case of lambda closure. The important note aboutthisis that it works not only as an implicit parameter, but also as a scope or context for all functions and objects called in a lexical scope where it is defined. Meaning that method`doBSomething`in fact has access to any public or private members of`A`as well as members of`B`itself.

Here comes Kotlin

Kotlin brings a completely new toy to the playground:extension functions. In Kotlin one can define an extension function like`A.doASomething()`which could be defined anywhere in the program, not just inside of`A`. Inside this function one has implicitthisparameter called receiver and pointing to the instance of`A`on which the method is called:

```
class A

fun A.doASomthing(){
    println("This extension method is called on $this")
}

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

Extension function do not have access to private members of its receiver, so it does not violate encapsulation.

The next important thing Kotlin brings on the table is a scoped code blocks. One can run an arbitrary code block using something as a receiver:

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

fun A.doASomthing(){}

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

In this example both functions could be called without additional`a.`at the beginning, because`with`function moves all code in the following block inside a*context*. Which means that all function in this block are called as if they are called on provided`a`object.

The final (at this moment) step to context-oriented programming is called member extension. In this case an extension function is defined inside another class, like this:

```
class B

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

fun main(){
    val a = A()
    val b = B()
    with(a){
        b.doBSomething() // this will work
    }
    b.doBSomething() // this will throw exception
}
```

The important thing is that`B`acquires some new behaviors, but only when called in specific parametric lexical context. The member extension function is equal member of class`A`, which means that the implementation of this function could be different depending on specific instance of`A`passed as a context. It could even interact with some state of`a`object.

Context-oriented dispatch

At the beginning of the article I've spent some time discussing different function dispatch approaches. The reason behind this preamble is that extension functions in Kotlin allow to define function dispatch in a new way. Now we can say that decision about which function to use is based not only on type of its parameters, but also on the lexical context where it is called. Meaning that the same expression in different contexts could have different meaning. Of course, from implementation point of view, nothing changed, we still have explicit receiver object which governs the dispatch for itself and other objects mentioned in its member extensions, but from point of view of syntax, the approach is different.

For example of how context-oriented approach is different from classic object-oriented one, let us consider classic Java problem of arithmetic operations on generic numbers. The`Number`class in Java and Kotlin is a parent class for all numbers, but contrary to specialized numbers like`Double`it does not define its own arithmetic operations. So one could not write something like:

```
val n: Number = 1.0

n + 1.0 // the `plus` operation is not defined on `Number`
```

The reason behind this is that there is no way to define arithmetic operations consistently on all type of numbers. For example Integer division is different from floating point division. In some special cases, user knows which type of operations he wants, but usually it does not make sense to define them globally. The object-oriented (and in fact functional) solution is to define a new type on top of our`Number`class, define operations for it and use it wherever it is needed (in Kotlin 1.3 it could be done by using inline classes). Instead let us define a context with those operations and apply it locally:

```
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)
}
```

In this case the calculation of`res`is done inside the context, which defines additional operations. The context is not necessary is defined locally, it could be passed implicitly as a function receiver. For example we can do it in a following way:

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

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

This means that the logic inside context is completely separated from the context implementation and could be written in different part of the program or even different module. In this simple example, context itself is a singleton without state, but it is possible to use stateful contexts.

Also one need to remember, that contexts could be nested:

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

Effectively combining the behaviors from two classes, but this feature currently is hard to control due to lack of multi-receiver extensions (KT-10468).

The power of explicit coroutines

Surprisingly, one of the best examples of context-oriented approach is already used in the language by coroutine library. The idea itself is explained in detail by Roman Elizarov in hisarticle. Here I would only like to highlight that`CoroutineScope`is a typical case of context-oriented design with stateful context.`CoroutineScope`plays two roles:

- It contains`CoroutineContext`which is necessary for running coroutines and is inherited when new coroutine is started.
- It contains the state of parent coroutine means to cancel parent coroutine if child one is failed.

Also, structured concurrency gives the best example of context oriented architecture:

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

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

Here`doSomeWork`is a context-based function defined outside its context.`launch`methods create two nested contexts which are equivalent to lexical scopes of corresponding functions (in this case both contexts have the same type, so inner context shadows outer one). The good starting point for someone, who wants to start to work with coroutines is theofficial guide.

DSL

There is a broad class of problems in Kotlin which are usually called DSL problems. Usually, by DSL people mean some code which provides a user-friendly builder to create some kind of complicated internal structure. It is not quite correct to call those builder DSLs since they use basic Kotlin syntax without any tweaks, but let's stick to common term.

DSL-builders in most cases are context oriented. For example if you want to create an HTML element you need first to check whether it is possible to add this particular element in this particular place.kotlinx.htmllibrary achieves it by providing context-based extensions of classes representing specific tags. In fact, the whole library is made of context extensions for existing DOM elements.

Another example isTornadoFXGUI builder. The whole scene graph builder is organized as a sequence of nested context-builder, where inner blocks are responsible for building children for outer blocks or adjusting parent properties. Here is an example from official documentation:

```
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()
                    }
                }
            }
        }
    }
}
```

Here lexical scope defines a context of its own (a quite logical one, since it represents a section of GUI and its internals) and has access to parent contexts.

What next: multiple receivers

The context-oriented programming provides a lot of tools for Kotlin developer and opens a new way to design code architecture, but do we need something more?

We probably do.

Current development in context-driven way is limited by the fact that one needs to define member extension in order to get some context-scoped behavior of class. It is OK, when it is a user-defined class, but what if we need to define some context-scoped behavior for a library class? Or if we want to create extension for already scoped behavior (for example add some extension inside`CoroutineScope`)? Currently Kotlin does not allow more then one receiver on an extension function. But multiple receivers could be added to language in a non-breaking backward-compatible way. The multiple receivers feature is currently being discussed (KT-10468) and will lead to aKEEPrequest in nearest future. The problem (or maybe the feature) of nested contexts is that it allows to cover most if not all use-cases oftype-classes, another sought-after possible feature. It is quite unprobable that both features will be implemented in the language at the same time.

Addendum

I would like to thank our friendly neighborhood Haskell-loving pedantAlexey Khudyakov for his remarks on the text and correction of my rather unconstrained usage of terms.