# Kotlin - functional programming

Donnerstag, 25. Juli 2024 18:28

**Author**: Damir Ljubic
**e-mail**: damirlj@yahoo.com

## Function signature

Kotlin as a language is Java sibling, and it's specially designed by JetBrains for Android development - to make this more pragmatic, especially in terms of the functional aspect of the language, treating functions as a first-class citizens.

Each language has "function" construct - as a block of code (instructions) that for a given inputs (if any) produces some result (if any), or alters the state of the program.
The signature of the regular function is

```
fun <name> (param1: Param1 ,…, paramN : ParamN) : <ReturnType> {
      // block of the code
}
```

For a simple function, we can even inline this, without using optional block of the code

```
fun createList(vararg args: Int): List<Int> = args.asList()
```

Well, this is yet another way to mimic in Kotlin already existing standard function - a factory method for List<T>: **listOf**(…), which is here clumsy used to demonstrate the flexibility of the function declaration in Kotlin.

For the functions that don't return the value, we can just skip it from declaration - or explicitly emphasize it with *Unit* - which has the same meaning as Java/C++ *void*.

## Function templates

As with other similar languages (like Java, and especially the C++), we can also specify the function templates

```
fun <Key,Value> printMap(map: Map<Key,Value>) {
  for ((key,value) in map) {
      println("($key, $value)")
  }
}
```

**Do you know the meaning of the higher-order function?**

This is language independent definition:
*The higher-order functions are the functions that either take other function(s) - callable(s) as argument(s), or return a function.*

This is especially useful for generic programming, where we want to implement *strategy design pattern* - to traversal through the collection - to iterate over the collection and apply the same functionality on each and every element, with one precondition: to satisfy the invocable concept (that has required signature).

In C++ std library, we have the entire <algorithm> written in this way - or to be precise, the entire std library is written fallowing this concept: having the collections detached from the algorithms.

Let's define the custom filter function (like the one in range library), that is a simple wrapper around the std::copy_if function

```
template <typename FwdIterator, typename UnaryPredicate>
auto filter (FwdIterator first, FwdIterator last, UnaryPredicate p) {
    using value_type = std::remove_cvref_t<decltype(*first)>;

    std::vector<value_type> out;
    out.reserve(std::distance(first, last));

    std::copy_if(first, last, std::back_inserter(out), p);
    return out;
}
```

In Kotlin, **we can't have a functions as template parameter** - we can't write

```
fun <Func, R> elapsed(f: Func):Pair<R, Long> {
    val start = System.currentTimeMillis()
    val result = f()
    return Pair(result, System.currentTimeMillis() - start)
}
```

We need to explicitly define the callback type

```
inline fun <R> elapsed(f: () -> R):Pair<R, Long> {
    val start = System.currentTimeMillis()
    val result = f()
    return Pair(result, System.currentTimeMillis() - start)
}
```

@note *inline* keyword here is hint to Compiler to inline the  higher-order function along with invoking function having it like in-place substitution at the calling site, where the callable will be also inlined, avoiding the calling overhead by creating the anonymous function object on the stack (that is - the same as in C++)

We can past the function arguments either by *reference*, or as *lambda*

```
fun someFunc() {}
val(result, time) = elapsed(::someFunc)// function reference

// or using the in-place callable - lambda
val(result, time) = elapsed {// lambda trailing syntax
    // block of code
}
```

Similar to Java/C++ the template parameter in Kotlin can be constrained - upper bounded to the base type

```
fun <T : Number> sumList(numbers: List<T>):Double {
    var sum = 0.0
    for(number in numbers) {
        sum += number.toDouble()
    }
    return sum
}
```

But also - it can be lower bound (*super* in Java), through something that is called use-site variance: using "in" keyword

```
fun <T> copy(from: Array<out T>, to: Array<in T>) {
    for (i in from.indices) {
        to[i] = from[i]
    }
}
```

In this example:

- Array<**out** T> means that the function can accept an array of T or any of its subtypes (covariance).
- Array<**in** T> means that the function can accept an array of T or any of its supertypes (contravariance).

Unlike C++, Kotlin don't have anything like *variadic template parameters* - parameter pack, which is required to specify the function of the arbitrary signature.
In Kotlin, we have *vararg* as the way to specify a function with arbitrary number of arguments - of the same type.
The closest way to get to the parameter pack is to use *Any* type - as super type for all non-null types (like *Object* in Java).

@note The ? beside the Any indicated that the arbitrary type can be a null-type as well

```
fun <R> elapsedAny(f: KFunction<R>, vararg args: Any?): Pair<R, Long> {
    return elapsed {f.call(*args) }
}
```

where the KFunction<R> is the *function interface* that employs the **reflection** mechanism - to determine the function type - signature at the runtime.

The *call* method - inspects whether the implementation of the function interface can be called with the given arguments

## Function membership

In C++ - since historically it is successor of C - we have free-functions, that are declared/defined outside of the class/structure scope.
In pure OOL like Java - the all functions are defined inside the custom type.
Interesting enough - Kotlin support free-functions (Top-Level Functions) as well.
Unlike C++ (before modules) - the ODR (One Definition Rule) issue is resolved in Kotlin (as well in Java) using the *packages* to avoid name conflicts and *import* mechanism at client side,
to explicitly specify the package - a namespace from which the functionality will be imported.

## Function visibility

If not other access modifier is specified - the default visibility for functions, as well for custom types and properties, is *public*.
That would mean - the function can be addressed from anywhere in the code.
On the other hand, *private* access modifier denotates the function which is visible only within the file or class where it's defined.
Similar to other OO languages, *protected* would mean accessible only to base (*open*) class and its derived subclasses - it's not applicable to free-functions.
There is Kotlin-specific *internal* visibility that refers to the accessibility within the Android module (app module, or, usually encapsulated within the same library module - to prevent name collisions of APIs from different modules).

## Extension Functions

are a convenient way to extend the class with some additional functionality, without involving inheritance - when you don't want to directly modify the class (like pure data classes, or sealed classes), or when the modification itself is not possible, since the class itself is not accessible (like being part of the external library).
This is also in line with Open-Closed Principle, where the new functionality is added, without literally modifying the original interface

The blueprint for these kind of functions is

```
fun <TypeYouWantToExtend>.functionName(args):ReturnType {
   //  function body
}
```

@note <TypeYouWantToExtend> is also known as *receiver*.
Inside the extension function, the receiver can be referenced with "this"

<Example>: https://pl.kotl.in/Th6BI8rEl

You may ask - what is a difference with free-functions taking a reference to the type being extended?

Well, there are certain **advantages** in the former approach:

- Clarity: it's more clear indication of the type being extended, since type is part of the function name, as well the intention of the function
- Composability: you can easily chain operations on the same object, instead of passing object explicitly as function argument

**Limitations**: The extension function can't access the private members of the type, and in case of the name collision - the type function will be called - shadowing the extension

## Infix notation

An interesting syntactic sugar in Kotlin is infix notation, where you can have more concise and therefore readable, binary operators-like expressions between instances of the same type.
For that, we use infix keyword on

- Member or extension functions with
- Single argument

## Local Functions

Interesting enough, Kotlin allows using local functions - functions defined within the scope of another - outer function, which works even with function templates, which is not the case in C++.
To mimic the local functions in C++, you need to either define anonymous function object ("closure"), or to define the local user type (which can't be class template), and then to call the member functions on this type.

# Code that demonstrates all these concepts
https://pl.kotl.in/FaxfR2HVp

## Suspendable vs Non-suspendable Function

This is story about the **coroutines**.
It's - as you can imagine, quite complex topic - so I'll try to simplify it.

Similar to the C++, the Kotlin coroutines are stackless - in C++ it means that
there is a *promise_type* interface that needs to be implemented by the coroutine return type - that will be allocated on the heap, to track the coroutine suspension/resume transitions - it's basically a state-machine.
(more on that: https://github.com/damirlj/modern_cpp_tutorials/blob/main/docs/C%2B%2B20/Coroutines.pdf)

In Kotlin, the coroutines are fairly simpler to use - without so many boilerplate code to implement.
To mark the function as the one that **has suspension point**: that can be suspend and resumed - there is a reserved word *suspend* to be used
@note It's similar to *co_await* and awaitable interface in C++

Let's rewrite our elapsed time measurement - to work for both - regular and suspend functions

```
private suspend fun <R> elapsed(f: suspend () ->R): Pair<R, Long> {
    val start = System.currentTimeMillis()
    val result = f() // suspension point
   return Pair(result, System.currentTimeMillis() - start)
}
```

@note In background - the compiler will create the Continuation object - to keep track about the state of the suspend function

The key components of the coroutines in Kotlin are:

- ☐ ***CoroutineScope*** class as a lifecycle manager of the coroutine
- ☐ ***CoroutineContext*** to define the *Dispatcher* - Scheduler (Default, IO, etc.), along some other thread attributes (name): as a thread context (actually, a thread pool) that will host the coroutine.
  There is a Job object as well - for handling the lifetime of the coroutine.

There are some predefined factory methods (builders) - for creating the coroutines on the fly

- ***runBlocking*** - factory method that creates coroutine that will block the calling thread, until the suspend function is not resumed.
  It's used to bridge the non-suspending (regular) function with the suspending one - providing the coroutine scope for its execution
- ***lunch*** - coroutine's factory method that returns the Job - a lifetime object of the coroutine on which one can join - wait on coroutine completion, cancel it (all pending jobs) - or even query the state of the coroutine (similar to *std::jthread* in C++).
  It can be used also in non-blocking way, without joining on completion - but rather detached from the calling thread - as fire-and-forget coroutine
- ***async***- factory method that creates deferred coroutine: returns *Deferred<T>* object - that can be lazily invoked: awaited on the result of the coroutine (similar to the *std::async* in C++)

```
/**
 * For measuring the elapsed time of a synchronous function.
 * It will be turned into blocking call - waiting on the function to be
 * executed within the coroutine context
 */
fun <R> func_elapsed(f: () ->R): Pair<R, Long> {
    val result : Pair<R, Long>
    runBlocking { // CoroutineScope
        result = elapsed(f)// elapsed() itself as suspension point
    }
    return result
}
```

**Source code**:
https://github.com/damirlj/modern_cpp_tutorials/tree/main/src/Kotlin/src/main/java/com/example/practice_kotlin
Unit test:
https://github.com/damirlj/modern_cpp_tutorials/blob/main/src/Kotlin/src/test/java/com/example/practice_kotlin/coroutines/TestCoroutines.kt

## Links

**#Scope functions**: https://kotlinlang.org/docs/scope-functions.html
**#Atomic Kotlin**: https://www.atomickotlin.com/