Fun with Kotlin

Duncan McGregor

@duncanmcg, www.oneeyedmen.com

@duncanmcg © Duncan McGregor 2019

What is a function?

— A block of code to perform a specific task

— Takes zero or more parameters, returns one result

— Mathematical functions, purity, referential transparency

— Methods, message sending, polymorphism

What fun?

- Methods
- Static methods
- Properties
- Top-level functions
- Extension functions
- Infix functions
- Operator functions
- Local functions

Can you have too much fun?

- Higher-order functions
- Function types
- Function values
- Lambda functions
- Inline functions
- Functions with receiver
- Function objects
- Implementing functions

Methods

Methods

are public and final by default

```
open class Person(val firstName: String, val lastName: String) {
    open fun fullName(): String {
        return firstName + " " + lastName
    }
}

class Employee(firstName: String, lastName: String) : Person(firstName, lastName) {
    override fun fullName(): String {
        return lastName + " " + firstName
    }
}
```

Methods

can be a single expression

```
class Person(val firstName: String, val lastName: String) {
    fun fullName(): String {
        return firstName + " " + lastName
    }
}

class Person(val firstName: String, val lastName: String) {
    fun fullName(): String = firstName + " " + lastName
}
```

Methods

can have an implicit return type

```
class Person(val firstName: String, val lastName: String) {
    fun fullName(): String = firstName + " " + lastName
}

class Person(val firstName: String, val lastName: String) {
    fun fullName() = firstName + " " + lastName
}
```

Methods

can be called with named arguments, or default parameters

```
class Person(val firstName: String, val lastName: String) {
    fun fullName(separator: String = " ") =
        firstName + separator + lastName
}

class PersonTests {
    @Test fun names() {
        assertEquals("Bob : TheBuilder", bob.fullName(" : "))
        assertEquals("Bob - TheBuilder", bob.fullName(separator = " - "))
        assertEquals("Bob TheBuilder", bob.fullName())
    }
}
```

Static Methods

are declared on a companion object

```
data class Person(val firstName: String, val lastName: String) {

    companion object {
        @JvmStatic fun parse(fullName: String): Person {
            val bits = fullName.split(" ")
            return Person(firstName = bits[0], lastName = bits[1])
        }
    }
}

class PersonTests {
    @Test fun `parse`() {
        assertEquals(
            Person("Bob", "TheBuilder"),
            Person.parse("Bob TheBuilder")
        )
    }
}
```

Properties

Properties

are functions really

```
data class Person(val firstName: String, val lastName: String) {
    val fullName: String
        get() {
            return firstName + " " + lastName
        }
}

class PersonTests {
    @Test
    fun names() {
        assertEquals("Bob TheBuilder", bob.fullName)
    }
}
```

Properties

can be single expressions

```
data class Person(val firstName: String, val lastName: String) {
    val fullName get() = firstName + " " + lastName
}

class PersonTests {
    @Test fun names() {
        assertEquals("Bob TheBuilder", bob.fullName)
    }
}
```

Properties

can be indexed

```
data class Person(val firstName: String, val lastName: String) {
    operator fun get(i: Int): String = when (i) {
        0 -> firstName
        1 -> lastName
        else -> throw IndexOutOfBoundsException("No name at index $i")
    }
}

class PersonTests {
    @Test fun names() {
        assertEquals("Bob", bob[0])
        assertEquals("TheBuilder", bob[1])
    }
}
```

Properties

can be resolved on construction

```
data class Person(
    val firstName: String,
    val lastName: String
) {
    val fullName = firstName + " " + lastName
}

class PersonTests {
    @Test fun names() {
        assertEquals("Bob", bob.firstName)
        assertEquals("TheBuilder", bob.lastName)
        assertEquals("Bob TheBuilder", bob.fullName)
    }
}
```

Properties

can be constant

```
data class Person(val firstName: String, val lastName: String) {
    val type = "Person"
}
```

Top-level Functions

Top-level Functions

have file scope

```
data class Person(val firstName: String, val lastName: String)

fun fullName(person: Person): String {
    return person.firstName + " " + person.lastName
}

class UtilityTests {
    @Test fun fullName() {
        assertEquals("Bob TheBuilder", fullName(bob))
    }
}
```

Top-level Functions

have the conveniences of methods

```
data class Person(val firstName: String, val lastName: String)

fun fullName(person: Person, separator: String = " ") =
    person.firstName + separator + person.lastName

class UtilityTests {
    @Test fun fullName() {
        assertEquals("Bob TheBuilder", fullName(bob))
        assertEquals("Bob : TheBuilder", fullName(bob, separator = " : "))
    }
}
```

Extension Functions

Extension Functions

'extend' a type

```
data class Person(val firstName: String, val lastName: String)

fun Person.fullName(separator: String = " ") =
    this.firstName + separator + this.lastName

class ExtensionTests {
    @Test fun fullName() {
        assertEquals("Bob : TheBuilder", bob.fullName(" : "))
        assertEquals("Bob TheBuilder", bob.fullName())
    }
}
```

Extension Functions

'this' is implied

```
fun Person.fullName(separator: String = " ") =
    this.firstName + separator + this.lastName

fun Person.fullName(separator: String = " ") =
    firstName + separator + lastName
```

Extension Functions

are statically resolved

```
open class Person(val firstName: String, val lastName: String)

class Employee(firstName: String, lastName: String) : Person(firstName, lastName)

fun Person.fullName(separator: String = " ") = "$firstName$separator$lastName"
fun Employee.fullName(separator: String = " ") = "$lastName$separator$firstName"

class ExtensionTests {
    @Test fun fullName() {
        val employeeBob = Employee("Bob", "TheBuilder")
        assertEquals("TheBuilder Bob", employeeBob.fullName())
        assertEquals("Bob TheBuilder", (employeeBob as Person).fullName())
    }
}
```

Extension Functions

can have other scopes

```
class ExtensionTests {

    private fun Person.fullName(separator: String = " ") =
        firstName + separator + lastName

    @Test fun fullName() {
        assertEquals("Bob : TheBuilder", bob.fullName(" : "))
    }
}
```

Extension Functions

are excellent for extending a type in a domain

```
fun toJsonNode(person: Person): ObjectNode {
    val result = objectMapper.createObjectNode()
    result.put("givenName", person.firstName)
    result.put("surname", person.lastName)
    return result
}

fun toPerson(objectNode: ObjectNode) = Person(
    objectNode["givenName"].textValue(),
    objectNode["surname"].textValue()
)

class MappingTests {
    @Test fun roundTrip() {
        assertEquals(bob, toPerson(toJsonNode(bob)))
    }
}
```

Extension Functions

are excellent for extending a type in a domain

```
fun Person.toJsonNode(): ObjectNode {
    val result = objectMapper.createObjectNode()
    result.put("givenName", firstName)
    result.put("surname", lastName)
    return result
}

fun ObjectNode.toPerson() = Person(
    this["givenName"].textValue(),
    this["surname"].textValue()
)

class MappingTests {
    @Test fun roundTrip() {
        assertEquals(bob, bob.toJsonNode().toPerson())
    }
}
```

Extension Functions

are excellent for chaining

```
fun toJsonString(objectNode: ObjectNode) =
    objectMapper.writeValueAsString(objectNode)

fun toJsonString(person: Person): String = toJsonString(toJsonNode(person))

fun ObjectNode.toJsonString() =
    objectMapper.writeValueAsString(this)

fun Person.toJsonString(): String = this.toJsonNode().toJsonString()
```

Extension Properties

are also a thing

```
val Person.fullName get() = firstName + " " + lastName

class ExtensionTests {
    @Test fun fullName() {
        assertEquals("Bob TheBuilder", bob.fullName)
    }
}
```

Infix Functions

Infix Functions

can be called without punctuation

```
data class Person(val firstName: String, val lastName: String)

infix fun String.the(occupation: String) = Person(this, "The$occupation")

class InfixTests {
    @Test fun the() {
        assertEquals(Person("Bob", "TheBuilder"), "Bob" the "Builder")
    }
}
```

Infix Functions

can be methods

```
data class Person(val firstName: String, val lastName: String) {
    infix fun and(another: Person) = setOf(this, another)
}

class InfixTests {
    @Test fun and() {
        assertEquals(
            setOf(Person("Bob", "TheBuilder"), Person("Muck", "TheTruck")),
            ("Bob" the "Builder") and ("Muck" the "Truck")
        )
    }
}
```

Infix Functions

are the gateway to operator overloading

```
data class Person(val firstName: String, val lastName: String)

infix operator fun Person.plus(another: Person) = setOf(this, another)

class InfixTests {
    @Test fun plus() {
        assertEquals(
            setOf(Person("Bob", "TheBuilder"), Person("Muck", "TheTruck")),
            ("Bob" the "Builder") + ("Muck" the "Truck") + ("Bob" the "Builder")
        )
    }
}
```

Local Functions

Local Functions

avoid scope pollution

```
fun complicatedThing(aString: String, anInt: Int): String {
    fun nitpickyDetail(aString: String, anInt: Int): String =
        aString + anInt

    //...

    return nitpickyDetail(aString, anInt)
}
```

Local Functions

close over parent scope

```
fun complicatedThing(aString: String, anInt: Int): String {
    fun nitpickyDetail(): String =
        aString + anInt

    //...

    return nitpickyDetail()
}
```

Higher-Order Functions

Higher-Order Functions

returning functions as values

```
fun occupier(occupation: String): (String) -> Person {
    return fun(firstName: String) = Person(firstName, "The$occupation")
}

class OccupierTests {
    @Test fun test() {
        val builderBuilder = occupier("Builder")
        assertEquals(Person("Bob", "TheBuilder"), builderBuilder("Bob"))
        assertEquals(Person("Brian", "TheBuilder"), builderBuilder("Brian"))
    }
}
```

Higher-Order Functions

taking functions as parameters

```
fun obfuscate(person: Person, f: (String) -> String) =
    Person(f(person.firstName), f(person.lastName))

class ObfuscationTests {
    @Test fun test() {
        fun translator(s: String): String = s.replace(vowels, "*")
        assertEquals(
            Person("B*b", "Th*B**ld*r"),
            obfuscate(bob, ::translator)
        )
    }
}
```

Higher-Order Functions

lambda expressions can replace functions

```
fun obfuscate(person: Person, f: (String) -> String) =
    Person(f(person.firstName), f(person.lastName))

class ObfuscationTests {
    @Test fun test() {
        assertEquals(
            Person("B*b", "Th*B**ld*r"),
            obfuscate(bob, { s: String -> s.replace(vowels, "*") })
        )
    }
}
```

Higher-Order Functions

'it' can be used for a single lambda parameter

```
fun obfuscate(person: Person, f: (String) -> String) =
    Person(f(person.firstName), f(person.lastName))

class ObfuscationTests {
    @Test fun test() {
        assertEquals(
            Person("B*b", "Th*B**ld*r"),
            obfuscate(bob, { it.replace(vowels, "*") })
        )
    }
}
```

Higher-Order Functions

lambdas can be moved outside the parens

```
fun obfuscate(person: Person, f: (String) -> String) =
    Person(f(person.firstName), f(person.lastName))

class ObfuscationTests {
    @Test fun test() {
        assertEquals(
            Person("B*b", "Th*B**ld*r"),
            obfuscate(bob) { it.replace(vowels, "*") }
        )
    }
}
```

Higher-Order Functions

and now obfuscate made an extension function

```
fun Person.obfuscatedBy(f: (String) -> String) =
    Person(f(firstName), f(lastName))

class ObfuscationTests {
    @Test fun test() {
        assertEquals(
            Person("B*b", "Th*B**ld*r"),
            bob.obfuscatedBy { it.replace(vowels, "*") }
        )
    }
}
```

Higher-Order Functions

or a little higher-order infix action

```
fun Person.obfuscatedBy(f: (String) -> String) =
    Person(f(firstName), f(lastName))

infix fun String.replacing(regex: Regex): (String) -> String = {
    it.replace(regex, this)
}

class ObfuscationTests {
    @Test fun test() {
        assertEquals(
            Person("B*b", "Th*B**ld*r"),
            bob.obfuscatedBy("*" replacing vowels)
        )
    }
}
```

Inline Functions

Inline Functions

can be used for control flow

```
fun Person.ifBlankFirstName(f: () -> String): Person =
    if (firstName.isNotBlank())
        this
    else
        this.copy(firstName = f())

class IfBlankFirstNameTests {
    @Test fun `returns a copy if first name is blank`() {
        assertEquals(
            Person("UNKNOWN", "Person"),
            Person("", "Person").ifBlankFirstName { "UNKNOWN" }
        )
    }
    @Test fun `returns receiver if first name not blank`() {
        assertEquals(
            bob,
            bob.ifBlankFirstName { "UNKNOWN" }
        )
    }
}
```

Inline Functions

can be used for control flow

```
inline fun Person.ifBlankFirstName(f: () -> String): Person =
    if (firstName.isNotBlank())
        this
    else
        this.copy(firstName = f())

class IfBlankFirstNameTests {
    @Test fun `early return`() {
        Person("", "Jamaflip").ifBlankFirstName {
            return
        }
        fail("did not return")
    }
}
```

Inline Functions

lots of nice built-ins

```
/**
 * Calls the specified function [block] with `this` value as its argument and returns its result.
 */
inline fun <T, R> T.let(block: (T) -> R): R {
    return block(this)
}

fun parse(fullName: String): Person {
    val bits = fullName.split(" ")
    return Person(firstName = bits[0], lastName = bits[1])
}

fun letParse(fullName: String) = fullName.split(" ").let { bits ->
    Person(firstName = bits[0], lastName = bits[1])
}
```

Inline Functions

lots of nice built-ins

```
fun printlnDebuggingBefore(person: Person) =
    doSomethingWith(person)

fun printlnDebuggingAfter(person: Person): Person {
    val result = doSomethingWith(person)
    println(result)
    return result
}
```

Inline Functions

lots of nice built-ins

```
/**
 * Calls the specified function [block] with `this` value
 * as its argument and returns `this` value.
 */
inline fun <T> T.also(block: (T) -> Unit): T {
    block(this)
    return this
}

fun printlnDebuggingBefore(person: Person) =
    doSomethingWith(person)

fun printlnDebuggingAfter(person: Person) =
    doSomethingWith(person).also { println(it) }
```

Inline Functions

lots of nice built-ins

```
fun Person.toJsonNode(): ObjectNode {
    val result = objectMapper.createObjectNode()
    result.put("givenName", firstName)
    result.put("surname", lastName)
    return result
}
```

Inline Functions

lots of nice built-ins

```
/**
 * Calls the specified function [block] with `this` value as its receiver and returns `this` value.
 */
inline fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}

fun Person.toJsonNode() = objectMapper.createObjectNode().apply {
    this.put("givenName", firstName)
    put("surname", lastName)
}
```

Implementing Function Types

Implementing Function Types

with lambdas

```
fun lambdas(people: Iterable<Person>) {
    val strings0: List<String> = people.map { person: Person -> person.toString() }
    val strings1: List<String> = people.map { person -> person.toString() }
    val strings2: List<String> = people.map { it.toString() }
}
```

Implementing Function Types

with values

```
fun lambdas(people: Iterable<Person>) {
    val lambdaValue: (Person) -> String = { it.lastName + " " + it.firstName }
    val strings0: List<String> = people.map(lambdaValue)

    val funValue = fun(person: Person) = person.firstName + " " + person.lastName
    val strings1: List<String> = people.map(funValue)
}
```

Implementing Function Types

with function and method references

```
fun converterFunction(person: Person) = person.firstName + " " + person.lastName

fun functionReferences(people: Iterable<Person>) {
    val strings0: List<String> = people.map(::converterFunction)
    val strings1: List<String> = people.map(Person::toString)
}
```

Implementing Function Types

with a class

```
class Converter(val separator: String): (Person) -> String {
    override fun invoke(person: Person): String = person.firstName + separator + person.lastName
}

fun classImplementingFunction(people: Iterable<Person>) {
    val converter = Converter(" : ")
    val strings0: List<String> = people.map(converter)
}
```

Implementing Function Types

with objects

```
fun objects(people: Iterable<Person>) {
    val anonymousConverterObject = object : (Person) -> String {
        override fun invoke(person: Person) = person.firstName + " " + person.lastName
    }
    val strings0: List<String> = people.map(anonymousConverterObject)

    val strings1: List<String> = people.map(TopLevelConverterObject)
}

object TopLevelConverterObject : (Person) -> String {
    override fun invoke(person: Person) = person.firstName + " " + person.lastName
}
```

Implementing Function Types

with a method

```
fun method(people: Iterable<Person>) {
    val converter = Converter(" : ")
    val strings0: List<String> = people.map(converter::convert)
}

class Converter(val separator: String) {
    fun convert(person: Person) = person.firstName + separator + person.lastName
}
```

Takeaway

- [Kotlin is not a strictly functional language]
- But support for first class functions is very good
- Allowing objects to implement function types allows productive mixing of OO and FP techniques
- Data oriented design is encouraged by extension functions, allowing us to pass around typed data and define operations in local contexts
- Features combine in interesting ways to allow terse but expressive code

That was fun?

— https://github.com/dmcg/fun-with-kotlin
— @duncanmcg
— www.oneeyedmen.com