GETTING STARTED WITH

Kotlin

BY SIMON WIRTZ, SOFTWARE DEVELOPER

CONTENTS

INTRODUCTION

Kotlin has become one of the most popular JVM languages in the past few months. One special reason is that it experienced a lot of attention in the Android community after Google made Kotlin an official language for Android development. Kotlin is being developed by JetBrains, who are responsible for the most famous IDEs out there, most notably IntelliJ IDEA. Nevertheless, it's an open source language, which can be found on GitHub.

The language is said to be very concise, safe in terms of error frequency, interoperable with Java and also offers many features that enable functional programming, writing type-safe DSLs and much more. Besides the JVM, Kotlin can compile for most Android versions, down to machine code using LLVM, and can also be transpiled to JavaScript. Kotlin has already been adopted in many popular frameworks and tools such as Spring and Gradle. It continues to gain traction in multiple domains, and there has never been a better time to **get started with Kotlin**.

WHERE TO START CODING

When you want to start writing your first Kotlin code there are quite a few ways to do that. Apparently, the recommended way is to work with IntelliJ IDEA, which offers the best support. As an alternative, one could also start with the command line or use JetBrains' Kotlin web IDE to do some Kotlin Koans. Whichever way you prefer, corresponding tutorials can be found here: kotlinlang.org/docs/tutorials.

BASIC SYNTAX

Kotlin was inspired by many modern programming languages like C#, Groovy, Scala and also Java. Even more, Kotlin can be seen as an extension to the Java language, making it better by adding functionality to existing standard classes (e.g. `String`, `List`) and of course by providing great features, which are in large part enabled by applying compiler-supported techniques. As in Java, Kotlin programs are entered via a main method, such as the following:

```
fun main(args: Array<String>): Unit {
    val inserted = "Kotlin"
    println("Let's get started with $inserted")
}
```

What we can see in this snippet is:

- **Functions** are initiated by the keyword `fun`, followed by a name
- **Parameters** and also **variables** in Kotlin are declared by defining a name and a type, both separated by a colon as you can see in `args: Array<String>`
- The return type of the `main` is `Unit`, also prefaced by a colon. In case of a `Unit` return, which corresponds to Java's `void`, the compiler does not require you to explicitly define the return type, so the part `: Unit` could be omitted
- Kotlin does not require you to use **semicolons** for separating statements (in most cases)
- **Type inference** is supported in many situations as shown with `val inserted`, which also could be declared with an explicit type as `val inserted: String`
- **String templates** can be used, which means that it's possible to include variables and even expressions in `Strings` directly using `$varname` or `${statement}` syntax
- `main` is declared without a wrapping `class` around it. Functions and variables in Kotlin may be declared at "top-level", i.e directly inside a package

- **No visibility** modifier is used here. Functions, classes, variables, etc. are `public` by default. When different visibility is needed, choose from:

Keyword	Effect on top-level declarations [1]	Effect on Class Members
`public`	visible everywhere	visible everywhere if class is accessible
`private`	visible inside the file only	visible inside the class only
`protected`	-	visible in class and subclasses
`internal`	visible inside the same module [2]	visible in the same module, if class is accessible

[1] Functions, properties and classes, objects and interfaces can be declared on the top-level.

[2] A module is a set of Kotlin files compiled together, such as an IntelliJ IDEA module, a Maven project, or a Gradle source set.

- Variables defined as `val` cannot be re-assigned, i.e. are read-only. Alternatively, if mutability is inevitable, `var` can be utilized, as shown in the next example:

```
var mutableVar = StringBuilder("first")
mutableVar = StringBuilder("second")
```

- Constructor is invoked without the `new` keyword, which is omitted from Kotlin

CONTROL FLOW: CONDITIONS

In Kotlin you can make use of `if`, `when`, `for`, and `while` for controlling the behavior of your code. Let's look at conditions first.

IF-STATEMENT

```
val min: Int
if (x < y) {
    min = x
} else {
    min = y
}
```

It's important to know that many **statements** in Kotlin can also be used as **expressions**, which, for instance, makes a ternary operator obsolete and apparently shortens the code in most cases:

```
val min = if (x < y) x else y
```

WHEN-STATEMENT

A `when` statement is very similar to `switch` operators and could, in theory, easily replace if-statements as they are much more powerful.

```
val y = when (x) {
    0 -> "is zero"
    1 -> "is one"
    2, 3 -> "two or three"
    is Int -> "is Int"
    is Double -> "is Double"
    in 0..100 -> "between 0 and 100"
    else -> "else block"
}
```

In a `when` statement, which can also be used as an expression, all branches are tried to match the input until one condition is satisfied. If no branch matches, the `else` is executed. As shown in the snippet, `when` branch conditions can be values, types, ranges, and more.

CONTROL FLOW: LOOPS

FOR-LOOP

In Kotlin, there's no conventional for-loop as you know it from C or Java. Instead, *foreach* loops are the default.

```
for (c in "charSequence") {
    //
}
```

In many cases, looping with an index is necessary, which can easily be achieved with the indices property that is defined for arrays, lists, and also `CharSequences` for example.

```
for (i in "charSequence".indices) {
   println("charSequence"[i])
}
```

Another way of iterating with `indices` is possible by using `withIndex()`.

```
for ((i,c) in "charSequence".withIndex()) {
    println("$i: $c")
}
```

Last but not least, Kotlin has ranges, which can also be utilized for indexed iterations as the following shows:

```
(0 .. "charSequence".length-1).forEach {
    print("charSequence"[it])
}
```

The range in this example is expressed with the common `..` syntax. To create a range which does not include the end element `(s.length)`, the `until` function is used: `(0 until s.length)`.

WHILE-LOOP

Constructs with `while` or `do-while` loops are straight-forward, all works as known from other common languages.

BASIC TYPES

In Kotlin, everything looks like an object to the user, even primitive types. This means that member functions can be called on every type, although some will be represented as *JVM primitives* at runtime.

NUMBERS

- The default number types are: `Double`, `Float`, `Long`, `Int`, `Short`, and `Byte`
- Underscores can be used to make large numbers more readable: `val million = 1_000_000`
- Number types offer conversion methods like `toByte(): Byte, toInt(): Int, toLong(): Long`
- Characters are no number type in Kotlin

CHARS

- A `Char` represents characters and cannot be treated as a number.
- They are declared within single quotes, e.g. `'4'`
- An explicit conversion from a `Char` to an `Int` can be accomplished with the `toInt()` method

BOOLEANS

- `Booleans` can have the two common values `true` and `false`
- They can be operated on with: `||`, `&&`, and `!`

STRINGS

- Strings are immutable sequences of characters.
- They offer an index operator `[ ]` for accessing characters at specified positions
- A string literal in Kotlin looks like `"Hello World"` or `"""Hello World with "another String" in it"""`
- The latter is called *raw* string that can contain any character without needing to escape special symbols
- `Strings` in Kotlin may contain template expressions

ARRAYS

- An array is represented by the class `Array`, which offers very useful methods to the client
- Values can be obtained via `get(index)` or `[index]`
- Values can be set via `set(index, value)` or `[index]=value`
- Arrays are invariant, i.e. an `Array<String>` cannot be assigned to a variable of type `Array<Any>`
- Special types for arrays of primitive types exist as `IntArray` or `ShortArray` for instance. Using those will reduce the boxing overhead.

CLASSES

A simple class can be declared like in this snippet:

```
class Person constructor(name: String) {}
```

The primary constructor is part of the class header, secondary constructors can be added in the class body. In the shown case, the `constructor` keyword could also be omitted, since it's only mandatory if you want to add annotations or visibility modifiers (default: public). Constructor parameters such as `name` can be used during the initialization of an object of this class. For this purpose, an `init` block would be necessary, because primary constructors can't contain code directly. Constructor arguments can also be used in property initializers that are declared in the class body, as shown here:

```
class Person(name: String, age: Int) {
    init {
        println("new Person $name will be born.")
    }
    val ageProp = age
}
```

As mentioned, Kotlin classes can contain properties, which are accessed by simply calling `obj.propertyName` to get a property's value and `obj.propertyName = "newValue"` to modify the value of *a mutable* (`var`) property. Declaring properties for classes can also be done in the primary constructor directly, which makes the code even more concise. Like in all methods, Kotlin supports **default parameters** for parameters, set with "`=`".

```
class Person(val name: String, val age: Int = 50)
```

Same as with local variables, instead of `val`, a property can be declared mutable using `var` instead. Note that you don't have to write an empty class body if no content is defined.

SPECIAL CLASSES

Besides ordinary classes, Kotlin knows a few special class declarations, which are worth knowing. The following will give a quick overview.

Type	Explanation
`data` class	Adds standard functionality for `toString`, `equals`, `hashCode` etc.
`sealed` class	Restricts class hierarchies to a set of subtypes. Useful with `when`
Nested class	Classes can be created in other classes, also known as "inner class"
`enum` class	Collect constants that can contain logic
`object` declarations	Used to create Singletons of a type

Of course, Kotlin also supports inheritance through `interfaces` and `abstract` classes.

FUNCTION TYPES AND LAMBDAS

In order to be able to understand idiomatic Kotlin code, it's essential to recognize how function types and especially lambdas look like. Just as you can declare variables of type `Int` or `String`, it's also possible to declare variables of function types, e.g. `(String) -> Boolean`.

```
val myFunction: (String) -> Boolean = { s ->
s.length > 3 }
myFunction("HelloWorld")
```

The variable is declared as a function type that takes a `String` argument and returns a `Boolean`. The method itself is defined as a lambda enclosed in curly braces. In the shown lambda, the `String` parameter is declared and named before the `->` symbol, whereas the body follows after it.

LAMBDA SPECIAL SYNTAX

The language designers decided on some special lambda features, which make them even more powerful.

1. `it`: implicit name of single parameters

   In many cases, lambdas are used with single parameters like in the previous example. In such situations, you don't have to give the parameter an explicit name. Instead, the implicit name `it` can be used.

   ```
   val myFunction: (String) -> Boolean = { it.length
   > 3 }
   myFunction("HelloWorld")
   ```

2. For unused parameters, use `_`

   In some cases, it might be unnecessary to make use of every possible available parameter in a lambda. The compiler warns the developer about such unused variables, which can be avoided by naming it with an **underscore**.

   ```
   val myFunction: (String, Int) -> Boolean = { s, _
   -> s.length > 3 }
   myFunction("HelloWorld", 42)
   ```

HIGHER-ORDER FUNCTIONS

If a function takes another function as an argument or returns another function as its result, it's called a **higher-order function**. Such functions are essential in Kotlin, as many library functions rely on this concept. Let's see an example:

```
fun main(args: Array<String>) {
    myHigherOrderFun(2, { it.length > 2 })
}

fun myHigherOrderFun(iterations: Int, test: (String) ->
Boolean){
    (0 until iterations).forEach {
        println("$it: ${test("myTestString")}")
    }
}
```

The function `myHigherOrderFun` defines two parameters, one of which is another function `test`. The function takes `test` and applies a `String` to it multiple times depending on what the first argument `iterations` is. By the way, the example uses a **range** to imitate an indexed `for` loop here.

The shown `main` function demonstrates the usage of a higher-order function by calling it with an anonymous function. The syntax looks a bit messy, which is why the language designers decided on a very important convention: If a lambda is the *last* argument to a function, it can be placed *after* the closing parentheses or, if it's the *only* argument, the parentheses can be omitted completely like shown with `forEach` above. The following snippet demonstrates this convention applied to an invocation of `myHigherOrderFun`.

```
//Lambda after closing parentheses
myHigherOrderFun(2) {
   it.length>2
}
```

TOP FEATURES

There are some features in Kotlin, everybody should be familiar with. These are essential for many libraries, standard functions and also advanced features like *Domain Specific Language* support.

NULL-SAFETY

The type system differentiates between nullable and non-null types. By default, a class like `String` cannot reference `null`, which raises the attention for `null`-related problems. As opposed to `String`, the type `String?` can hold `null`. This does not make a big difference on its own. Therefore, working with nullable types implies having to handle nullable values in a special way.

```
var b: String? = "couldBeNull"
b = null //okay

// 1. Access directly: does not compile, could throw NPE
// val len = b.length
```

code continued on next page

```
//2. Use safe-operator
val len = b?.length

//3. Check nullability before accessing
if(b != null){
   b.length
}
```

It's possible to check whether a variable is not `null` before accessing it. In such cases, the compiler permits the usage without special safety measures. Alternatively, `b?.length` expresses: call `length` on `b` if it's not `null`, otherwise the expression returns `null`. The return is of type `Int?` because `null` may be returned. Chaining such calls is possible, which is very useful. Other operators used with nullable types are shown in the following overview.

Operator	Use case	Example
`!!`	Ignore warnings of compiler and overcome null checks. Use cautiously.	`val x: String? = "nullable" x!!.length`
`?:`	The **elvis operator** is used to give an alternative for `null` results.	`val x: String? = "nullable" val len: Int = b?.length ?: 0`
`as?`	A **safe cast** tries to cast a variable in a type and results in `null` if the cast is not possible.	`val i: Int? = s as? Int`

EXTENSIONS

Another essential feature of Kotlin is **extensions**. An extension is used to extend a class with new functionality without having to inherit from that class. Extensions can have the form of properties and functions. The Kotlin standard library contains a lot of such extensions, like the following defined on `String`:

```
public fun String.substring(range: IntRange): String =
substring(range.start, range.endInclusive + 1)

//usage
"myString".substring(0..3)
```

In this example `String` is the **receiver** of the defined `substring(range: IntRange)` function. An extension function can use visible members of its receiver without additional qualifiers since `this` refers to the receiver. In the snippet, `String`'s standard method `substring(startIndex: Int, endIndex: Int)` is called in that way. The extension is called on a `String` as if it was a regular method.

It's also possible to extend a class with *properties*. For example, `Int` can be extended with a property that represents its version of `BigDecimal`. This might be useful if otherwise, the constructor of `BigDecimal` had to be used too many times.

```
val Int.bd
    get() = BigDecimal(this)

val bd: BigDecimal = 5.bd
```

Extensions are mostly defined on top-level and can be used in other files after they have been imported explicitly.

LAMBDA WITH RECEIVER

Higher-order functions can be even more powerful if used with "lambdas with receiver". It's possible to call function literals with a specific receiver object, similar to the extension functions. As a result, members of the receiver can directly be accessed inside the lambda without having to use additional qualifiers. This feature is the foundation for Kotlin's fantastic support for writing *Type-Safe Builders*, also known as *Domain Specific Languages*.

```
fun <T> T.apply(block: T.() -> Unit): T {
    block()
    return this
}
```

This snippet shows a slightly simplified version of the `apply` function, which is part of Kotlin's standard library. It's an extension function on the generic type `T`, and thus can be used with any object. The function takes a function literal with `T` as its receiver and executes the block before `this` (the receiver of `apply`) is being returned.

```
data class GuiContainer(var width: Int = 0, var height:
Int = 0, var background: String = "red") {
    fun printMe() = println(this)
}

fun main(args: Array<String>) {
    val container = GuiContainer().apply {
        width = 10
        height = 20
        background = "blueish"
        printMe()
    }
}
```

In this example, the data class `GuiContainer` is created with default parameters and then the `apply` method is called on it. It's possible to set mutable properties and call methods of the receiver `GuiContainer` like shown with the invocation of `printMe()` at the end. Since `apply` returns the receiver after it completes, it can directly be assigned to a variable.

IDIOMATIC KOTLIN

Kotlin tries to encourage particular *coding idioms* to be used. These are partially listed in the documentation and also in some

community driven articles. The following will present some of these idioms by example.

1. Use `when` as an expression if possible

```
fun analyzeType(obj: Any) =
    when(obj){
        is String -> "is String"
        else -> "no String"
    }
```

2. Use **elvis** operator with `throw` and `return` to handle nullable values

```
class Person(val name: String?, val age: Int?)

fun process(person: Person) {
    val pName = person.name ?: throw
IllegalArgumentException("Name must be provided.")
    println("processing $pName")
    val pAge = person.age ?: return
    println("$pName is $pAge years old")
}
```

3. Make use of **range** checks

```
fun inLatinAlphabet(char: Char) = char in 'A'..'Z'
```

4. Prefer **default parameters** to function overloads

```
fun greet(person: Person, printAge: Boolean = false) {
    println("Hello ${person.name}")
    if (printAge)
        println("${person.name} is ${person.age} years
old")
}
```

5. Use **type aliases** for function types

```
typealias StringPredicate = (String) -> Boolean

val pred: StringPredicate = {it.length > 3}
```

6. Use **data** classes for multiple return values

```
data class Multi(val s: String, val i: Int)

fun foo() = Multi("one", 1)

fun main(args: Array<String>){
    val (name, num) = foo()
}
```

7. Prefer **extension** functions to utility-style functions

```
fun Person.greet(printAge: Boolean = false) {
    println("Hello $name")
    if (printAge)
        println("$name is $age years old")
}
```

8. Use **apply** for object initialization

```
data class GuiContainer(var width: Int = 0, var height:
 Int = 0, var background: String = "red") {
    fun printMe() = println(this)
}

fun main(args: Array<String>) {
    val container = GuiContainer().apply {
        width = 10
        height = 20
        background = "blueish"
        printMe()
    }
}
```

9. Use `compareBy` for complex comparisons

```
fun sort(persons: List<Person>): List<Person> =
        persons.sortedWith(compareBy(Person::name,
Person::age))
```

10. Use `mapNotNull` to combine map and filter for non-null values

```
fun getPersonNames(persons: List<Person>): List<String> =
    persons.mapNotNull { it.name }
```

11. Use `object` to apply Singleton pattern

```
object PersonRepository{
    fun save(p: Person){}
    //...
}

//usage
val p = Person("Paul", 40)
PersonRepository.save(p)
```

12. Do not make use of `!!`

```
//Do not use !!, there's always a better solution
person!!.address!!.street!!.length
```

13. Prefer read-only data structures

```
//Whenever possible, do not use mutable Data Structures

val mutableList: MutableList<Int> = mutableListOf(1, 2,
3)
mutableList[0] = 0

val readOnly: List<Int> = listOf(1, 2, 3)
readOnly[0] = 0 // Does not compile
```

14. Use `let` to execute code if receiver is *not null*

```
fun letPerson(p: Person?) {
    p?.let {
        println("Person is not null")
    }
}
```

RESOURCES

LANGUAGE REFERENCES

- Official Reference Documentation
  kotlinlang.org/docs/reference
- GitHub repository
  github.com/JetBrains/kotlin
- Collection of Tools and Frameworks
  https://kotlin.link
- Operators and Keywords Overview
  https://kotlinlang.org/docs/reference/keyword-reference.html

COMMUNITY

- Slack
  kotlinlang.slack.com
- Twitter
  twitter.com/kotlin
- Newsletter
  kotlinweekly.net
- Discussion
  discuss.kotlinlang.org

BLOGS

- JetBrains
  blog.jetbrains.com/kotlin/
- Simon Wirtz
  blog.simon-wirtz.de

MISC

- Books
  kotlinlang.org/docs/books.html
- Online IDE
  try.kotlinlang.org

Written by Simon Wirtz, Software Developer

I'm having fun as a Software Developer with a focus on Java (web-)applications for almost 6 years now. Currently I work for a German software company in very different areas, mostly involved in Spring and OSGi development with Java. I started supporting Kotlin in January 2017 by blogging and speaking about it and I also use the language whenever possible. I can be found on Twitter @s1m0nw1.

DZone, Inc.
150 Preston Executive Dr. Cary, NC 27513
888.678.0399 919.678.0300