Kotlin срещу Scala – сравнителен анализ

Петко Данов

Резюме: *Представен само преди няколко години, Kotlin е един от многото езици, които използват виртуалната машина на Java, но в този кратък период придобива популярност. Scala е чист обектно-ориентиран език за програмиране, обединяващ в себе си предимствата на високо мащабируемите функционални езици за програмиране. Целта на настоящия документ е да сравни структурно двата езика и да идентифицира предимствата и недостатъците на всеки от тях.*
Ключови думи: *Котлин, Скала, Програмни езици*

Kotlin vs Scala – a comparative study

Petko Danov

Abstract*: Introduced just a few years ago, Kotlin is one of many languages that use the Java virtual machine, but it is gaining popularity in this short period. Scala is a pure object-oriented programming language that combines the advantages of highly scalable functional programming languages. The purpose of this paper is to compare the two languages structurally and to identify the advantages and disadvantages of each.*
Key words: *Kotlin; Scala; Programming languages.*

1. INTRODUCTION

The development of the Kotlin programming language was started in 2010 by Jet Brains. The first stable version was released in February 2016, and in May 2017, Google included Kotlin in Android Studio 3.0. Although it uses a JVM, it can also be compiled into JavaScript and machine code [1]. Kotlin is a static-typed object-oriented language. Kotlin can be used both in an object-oriented programming style and in a functional or mix of both styles [2]. For example, in Kotlin it is possible to declare functions outside the classes. Kotlin supports non-nullable types, which makes applications less susceptible to null point dereference (NullPointerException). Support for smart casting, higher-order functions and extension functions is also available.

The development of the Scala programming language began in 2001 as an academic project developed at the Ecole Polytechnique Federale de Lausanne (EPFL) by Martin Oderski and his team [3]. The first stable version was introduced in 2003, and the idea of the language's creators was to combine functional and object-oriented programming.

Scala is a multiparadigm programming language that supports both object-oriented and functional programming approaches. Scala is scalable, suitable for everything from short scripts to large-scale component-based applications [4]. Scala uses the so-called "actor model" to maintain modern parallelism. This programming language also supports the so-called "lazy evaluation". With the inclusion of macros, Scala is also distinguished by the creation of complex internal domain-specific DSL languages. Scala also supports first-order operations and allows the use of nested functions.

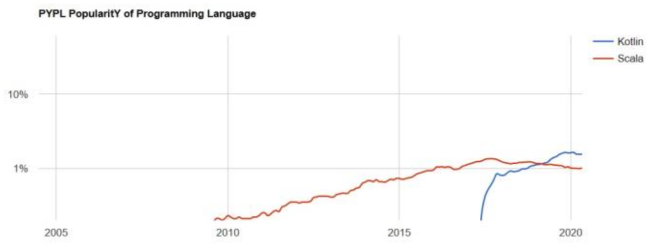

Fig. 1. *The popularity of Scala and Kotlin according to PYPL.*

Both Kotlin and Scala can use the Java Virtual Machine (JVM) to execute programs before compiling the code to bytecode. Several hundred JVM programming languages are available, including Groovy, Clojure, Ceylon, Xtend, Fantom, X10, according to [5]. Some of the most popular among them, according to [6] and [7], are Kotlin and Scala.

The difference in popularity between these two programming languages according to [8] can be seen in fig. 1.

2. DATA TYPES IN KOTLIN AND SCALA

The basic building blocks of programming languages are data types. The primitive data types are embedded in the language, while the abstract, system or user-defined data types are not part of the language, but realized as libraries, packages or modules.

Both Kotlin and Scala support the following three basic types of data:

- literal or literal constant;
- named constant or symbolic constant;
- variable.

The variety of data supported by the two programming languages can be seen in Table 1.

Table 1. Data types in Kotlin and Scala

	Kotlin	**Scala**
Numeric (integer)	Byte (8 bits) Short (16 bits) Int (32 bits) Long (64 bits)	Byte (8 bits) Short (16 bits) Int (32 bits) Long (64 bits)
Numeric (floating point)	Float (32 bits) Double (64 bits)	Float (32 bits) Double (64 bits)
Boolean	Boolean	Boolean
Alphanumeric (character)	Char (16 bits)	Char (16 bits)
Alphanumeric (string)	String	String
Null object	null	
Base class	Any	Any

Primitive data types are objects in Kotlin and Scala. In Kotlin, some of the types may have a special internal representation - for example, numeric, single character and Boolean types can be represented as primitive values at runtime - but to the user they look like ordinary classes [2].

The base class in Kotlin and Scala is Any. Types in Kotlin are divided into nullable and non-nullable. In Kotlin Any is a super type for all types but it cannot contain a value of null, the Any? type should be used if that is necessary. In Scala, Any subclasses fall into two categories: value classes that inherit from scala.AnyVal and reference classes that inherit from scala.AnyRef. For each primitive Java data type, Scala has a corresponding value class that is predefined [9].

3. OPERATIONS AND EXPRESSIONS

Operations represent the actions that are performed on operands when calculating expressions. Table 2 is completed using data from [10], [2], [11], [12], [13] and [14].

Table 2. Data comparison of operations and expressions in Kotlin and Scala

Scope resolution	**Kotlin**	**Scala**
Parenthesis	()	()
Function call	()	()
Array subscript	[]	()
Access via object	., ?.	.
Post increment	++	
Post decrement	--	
Unary plus, minus	+, -	+, -
Logical negation	!	!
Casting operator	as (type)	asInstanceOf[type]
Creating object		new
Multiplicative	*, /, %	*, /, %
Additive	+, -	+, -
Shift left, shift right	shl(bits), shr(bits)	<<, >>
Right with 0 ext	ushr(bits)	>>>
Relational	<, <=, >, >=	<, <=, >, >=
Testing object type	is (type)	isInstanceOf
Equality	==, !=, ===, !==	==, !=
Bitwise AND	and(bits)	&
Bitwise excl OR	xor(bits)	^
Bitwise OR	or(bits)	|
Boolean AND	&&	&&
Boolean OR	||	||
Ternary		

condition		
Assignment	=	=
Compound assignment	+=, -=, *=, /=, %=	+=, -=, *=, /=, %=

As mentioned above, the types in Kotlin are divided into nullable and non-nullable. This is one of the most important differences between this language and Scala - Kotlin's explicit support for the so-called nullable types. Placing a question mark after the type name explicitly allows the variable to contain null. Similarly, when accessing an object when it is of nullable types, (?.) Is used instead of (.).

It should be emphasized that, unlike Kotlin, in Scala every operation is a method. Each method of one parameter can be used as an infecting operator. For example + can be called with dot notation.

As can be seen in Scala, unlike Kotlin, there are no post-incrementing and post-decrementing operators. In Scala, this can be achieved by using binary operators (+=, -=) to increase and decrease an integer.

It should also be noted that the operator used to cast types in Kotlin is as, and in Scala is asInstanceOf. In Kotlin, the is operator checks whether an expression is an instance of a class, while in Scala, the isInstanceOf operator is used for this purpose.

It should be pointed out that Kotlin does not use the new keyword to create a new object, unlike Scala, where using it allocates memory for the newly created object. In Kotlin, an object is created by calling the constructor, like any ordinary function in a language.

Kotlin provides several functions (in infix form) for performing bitwise and bitshifting operations. Bitwise and bitshifting operators for bit-level operations are used for only two types – Int and Long. For comparison, bit-level operations in Scala are supported for the byte, short, int, and long types.

There is no special ternary operator in both Kotlin and Scala. For this purpose, the if / else expression can be used in both programming languages, which can return one of two values under a certain condition.

In Kotlin, there are two types of equality: structural equality (check for equals ()) and reference equality (two references point to the same object) [2]. Scala also implements the equals () method and the == operator, but in this programming language, structural equality is checked in these two ways. The eq and ne methods, respectively, are used to check the reference equality in Scala. However, it should be emphasized that the structural equality check in Kotlin is performed by the operator == (respectively, for a difference check != is used) and the reference equality check – by the operator === (respectively, for a difference check !== is used).

4. MAIN STATEMENTS IN KOTLIN AND SCALA

Kotlin and Scala support both declarative and executable statements.

The declarative statements are used to explicitly declare the data before it's used. Sample declarations can be found in Table 3:

Table 3. Examples of declarative statements in Kotlin and Scala

Kotlin	**Scala**
Declaring a variable	
var name: String	var name: String
Declaring a variable with initialization	
var name = "Kotlin" var name1: String = " Kotlin example"	var name = "Scala" var name1: String = "Scala example"
Declaring a named constant	
val i: Int	val i: Int

As can be seen in the example above - when declaring a variable in Kotlin and Scala, the implementation starts with a keyword, and when declaring a variable with initialization, specifying the type is not necessary, but it is possible to specify it explicitly. In such a case, the compiler parses the initialized expression and uses its type as the variable type. The keywords used to declare a variable are two - var and val. The value of a variable declared with var can be changed, unlike a variable declared with val, the value of which becomes an invariant reference after initialization.

Executable operators include an assignment operator, conditional branch operators and cyclic repeat operators. Three control structures are known at the level of execution of a program operator: sequence, selection, repetition, iteration [15], which correspond to the types of algorithms - linear, branched and cyclic, to which the executable operators conform. Table 4, denoting the syntax of an assignment operator, a composite operator and a branch operator, is compiled using data from [10], [2], [11], [12], [13] and [14].

Table 4. Examples of executable statements in Kotlin and Scala

Kotlin	**Scala**
Assignment statement	
<variable> = <expression>	<variable> =<expression>
Composite statement	

It is formed by a group of operators enclosed by curly brackets	It is formed by a group of operators enclosed by curly brackets
Branching statements	
if (<condition>) <statements>	if (<condition>){ <statements> }
if (<condition>){ <statements> } else { <statements> }	if (<condition>){ <statements> } else { <statements> }
if (<condition>){ <statements> else if(cond_n){ <statements> }else{ <statements> }	if (<condition>){ <statements> else if(cond_n){ <statements> }else{ <statements> }
when(<intExpression> { <value1> -> <statements> <valueN> -> <statements> else -> <statements> }	<intExpression> match{ case <value1> =><statements> case <valueN> =><statements> }
Repetition statements	
while(<condition>){ <statements> }	while(<condition>){ <statements> }
	for(<identifier> <- condition>){ <statements> }
do { <statements> } while(<condition>)	do { <statements> } while(<condition>)
Repetition statement based on data structures	
for (item in collection) { < statements > }	for(<identifier> <- <array>) if <condition1>; if<conditionN>){ < statements > }

Kotlin and Scala support traditional constructions with the if, while, and do-while statements. It should be noted here that if can be used not only as a branch operator but also as an expression. As already mentioned - in Kotlin and Scala there is no ternary operator, but its functionality is achieved by using the expression if-else.

The when branch operator in Kotlin can be considered as an extended version of the switch operator in Java. To the left of "->" is an expression called a condition. It is also possible to have a comma-separated list of expressions instead of one expression. If the argument given to when is equal to at least one of these expressions, the body of this condition is satisfied and the expression or block specified to the right of the "->" character is executed. The conditions are checked sequentially, from top to bottom. If none of the conditions are met, the entry after else is executed.

Similar possibilities are implemented in Scala with the use of a match expression, which can be used to choose between several possible alternatives, similar to the use of several if statements. The first pattern that matches is selected, and the part following the arrow is selected and executed. Furthermore, Scala match expressions can return a value.

Kotlin does not have the traditional for-loop, which initializes a variable whose value is updated at each step of the iteration and terminates when a certain value is reached. Kotlin uses a data structure for-loop. In this way, any data structure that provides an iterator (Range, Array, String, Collection) can be traversed. The concept of range is the interval between two values - start and end.

In Scala, where the for-cycle is also known as for-comprehensions, different forms / variants of the for-cycle are supported. For example, the operator "<-", called a generator, is used to generate values in a certain range, containing initial and final values. Like the foreach loop in Scala, the for-loop can also be used to crawl elements of collections, allowing filtering of certain elements of the collection using one or more conditional if statements. The construction of a for-loop using the yield keyword allows the return of a new collection of the same type.

5. SUBROUTINES IN KOTLIN AND SCALA

Subroutines in Kotlin are called functions, and Scala has both functions and methods. Function declarations in Kotlin can be combined in namespaces and a function can be declared in a namespace. In addition to the top level in a file, functions can be declared as local, as member functions or extension functions.

Scala methods are part of a class that has a name, a signature, while functions are objects that can be assigned to variables. The methods in Scala can be converted to a function and for this purpose the corresponding method must be called using "_" after its name. A function can be declared anywhere in the source file, and nested function definitions are possible in Scala, i.e. function definitions in other function definitions. Scala has a number of traits for representing functions with different numbers of arguments. As an instance of a class that implements any of these features, an object function can have methods.

The main differences in the implementation of subroutines in Kotlin and Scala can be seen in Table 5.

Table 5. Differences in implementation of subroutines in Kotlin and Scala

	Kotlin	**Scala**
Default Argument	Default values can provide values to parameters in function definition	Default values can provide values to parameters in function definition
Named argument	Function parameters can be named when calling functions.	Function parameters can be named when calling functions.
Extension Functions	Extensions can be created by prefixing the name of a class to the name of the new function, so an existing class can be extended with new functionality.	-
Higher-Order Functions and Lambdas	Comes as one of the prebuilt features.	Higher order functions take other functions as parameters or return a function as a result. This is possible because functions are first-class values in Scala.
Tail recursive functions	When a recursive function is marked with the tailrec modifier, the compiler optimizes the recursion, without the risk of stack overflow.	Only directly recursive calls to the current function are optimized. One can require that a function is tail-recursive using a @tailrec annotation.
Inline Functions	Inline function marked, as well as function parameters.	
Type of variables	Variables are by default mutable type.	Variables are by default mutable type.

6. CLASSES AND OBJECTS IN KOTLIN AND SCALA

In Kotlin and Scala, objects are created as an instance of the class using constructors.

In Kotlin, however, there is a difference between a primary constructor (declared outside the class body) and secondary constructors (declared in the class body). The primary constructor initializes the class, while the secondary constructors help to include additional logic.

In Scala, the main constructor represents the body of the class and its list of parameters described immediately after the class name, while the auxiliary constructors are declared in the body of the class. An auxiliary constructor must, in the first place in its body, call either another auxiliary constructor or the main constructor.

There are no Static Members in either Kotlin or Scala. In these two programming languages it is necessary to use companion object for this purpose. An object declaration inside a class can be marked with the companion keyword.

Kotlin inner classes do not have access to the outer class instance unless explicitly requested by adding the inner modifier. In Scala, an inner class cannot access the outer class instance.

In Kotlin, Data classes are used to define classes that store only properties. These classes do not contain any methods, only properties and, unlike an ordinary class, have no body. Because Scala supports so-called "Case Classes", they can be used to model immutable data. The case classes are ordinary classes that export their constructor parameters and that provide a recursive decomposition mechanism through the so-called "Pattern matching". The parameters of the case class constructor are treated as public values and can be accessed directly.

Kotlin supports delegation natively by language in the form of Class Delegation and Delegated Properties. The Delegation class uses the Delegation pattern as an alternative to inheritance and allows multiple inheritance to be implemented. Scala, on the other hand, does not support multiple inheritance, but allows multiple traits to be inherited.

7. CONCLUSION

Our study examined the main structural differences between the two programming languages. It can be concluded that to a large extent Kotlin offers features such as Smart casts, Extension functions, Delegation, Null-safety. With a short and intuitive syntax, Kotlin can increase the productivity of the programmer.

Nevertheless, Kotlin does not offer anything fundamentally new.

Scala offers features such as full support for pattern matching, Domain specific language (DSL) support. It offers concise notation and provides code complexity optimization. Scala is less readable because of nested code and is not backward compatible.

Each of these two programming languages has its own features. It is up to the developers to choose the development language that is most suitable for their needs.

ACKNOWLEDGMENT:

The work presented in the paper is supported by research grant 202ПД0010-09 financed by the Research and Development Sector of TU-Sofia.

REFERENCES:

[1] D. Jemerov, Kotlin 1.2 released: sharing code between platforms, available online at: https://blog.jetbrains.com/kotlin/2017/11/kotlin-1-2-released

[2] Kotlin Language Documentation, available online at: https://kotlinlang.org/docs/kotlin-docs.pdf

[3] M. Odersky, Scala's prehistory, available online at: https://www.scala-lang.org/old/node/239.html,

[4] D. Wampler & A. Payne, Programming Scala: scalability = functional programming + objects, O'Reilly media, ISBN: 1491950153

[5] R. Urma, Alternative languages for the JVM. A look at eight features from eight JVM languages, oracle.com, available online at: https://www.oracle.com/

[6] IEEE Spectrum Interactive, The Top Programming Languages, available online at: https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019

[7] TIOBE – The Software Quality Company, TIOBE Index for December 2019, available online at: https://www.test.tiobe.com/tiobe-index/

[8] PYPL PopularitY of Programming Language, Worldwide, May 2020 compared to a year ago, available online at: http://pypl.github.io/PYPL.html

[9] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, An overview of the Scala programming language, EPFL Lausanne, Switzerland, 2004, IC/2004/64

[10] S. Bonev, Comparative study of programming languages („Сравнително изучаване на езици за програмиране“), St. St. Cyril and Methodius National Library, Sofia 2018, ISBN: 978-619-91041-1

[11] D.Jemerov and S. Isakova, Kotlin in action, Manning publications, 1 edition (February 19, 2017), ISBN: 978-1617293290

[12] S. Samuel and S. Bocutiu, Programming kotlin, Packt Publishing, Birmingham 2017, ISBN 978-1-78712-636-7

[13] M. Odersky, L. Spoon, B. Venners, Programming in scala, Artima press, Mountain View, California, 2008, ISBN: 978-0-9815316-1-8

[14] M. Odersky, The scala language specification version 2.9, Programming methods laboratory, EPFL, Switzerland, available online at: https://www.scala-lang.org/old/sites/default/files/linuxsoft_archives/docu/files/ScalaReference.pdf

[15] S. Bonev, Programming technology (Технология на програмирането), Siela, Sofia 2000, ISBN: 9546492981

About the authors:

Petko, Danov, Department of Computer Systems, Faculty of Computer Systems and Technologies, Technical University of Sofia, danov@tu-sofia.bg