Comparative study Java vs Kotlin

Daniela Gotseva, Yavor Tomov and Petko Danov
Department of Computer Systems, Faculty of Computer Systems and Technologies
Technical University of Sofia
8 Kliment Ohridski blvd., 1000 Sofia, Bulgaria
dgoceva@tu-sofia.bg, yavor_tomov@tu-sofia.bg, danov@tu-sofia.bg

Abstract – **Among the great variety of programming languages, Java stands out vividly, becoming, for several decades, one of the most preferred languages for programmers. Introduced only a few years ago, Kotlin is one of the many languages that use the Java Virtual Machine, but in this short period it has gained popularity. The purpose of this paper is to compare the two languages structurally and to identify the advantages and disadvantages of each.**

Keywords – **Kotlin; Java; Programming languages.**

I. INTRODUCTION

Since its official introduction almost a quarter-century ago, Java is now one of the most popular programming languages in the world, according to [1] and [2]. Probably one of the main reasons contributing to this fact is its platform independence - the code is compiled to byte code and the program is executed via Java Virtual Machine (JVM) [3]. This provides easy portability between different software or hardware platforms, i.e. through a virtual machine, a Java application can run independently of the architecture or operating system used by a computer.

Could one of Java's strongest points be the reason for the dramatic decline in popularity of this programming language - according to [4] several hundred JVM programming languages are available, including Groovy, Clojure, Ceylon, Xtend, Fantom, X10, Scala and Kotlin. Of particular note is the ever-growing in popularity Kotlin language, which has been declared a priority programming language for Android application development [5].

Kotlin's development began 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 JVM, it can also be compiled into JavaScript and machine code.

Like Java, Kotlin is a static-typed language. Kotlin is also an object-oriented language, but unlike Java (no support until Java 8), it also supports functional constructions. Kotlin can be used both in object-oriented and in functional programming style or in a mix of both styles [6]. For example, in Kotlin it is possible to declare functions outside the classes. In Java, static methods are used for these purposes, resulting in classes whose instances are never created but instead only static methods are called.

Kotlin supports non-nullable types, making applications less susceptible to null point dereference (NullPointerException). Smart casting, higher-order functions and extension functions support is also available. Unlike Java, in Kotlin there are no checked exceptions.

II. DATA TYPES

Data types are the basic building blocks of programming languages. 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 Java and Kotlin support the following three basic types of data:

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

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

TABLE 1. DATA TYPES IN JAVA AND KOTLIN

	Java	Kotlin
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	null
Base class	Object	Any

Primitive data types are not Java objects, unlike the String and Object reference types. The main difference between primitive and reference data types is how they are organized in the memory. Primitive data types are stored on the stack, they exist only in their field of view. In reference types, an address (reference) is stored in the stack, in the heap of which is the object itself.

Java primitive types cannot be used as generic types because they do not support method calls and cannot obtain a null value. For each primitive type a so-called wrapper class exists which removes these restrictions.

Primitive data types are objects in Kotlin. Some types may have special internal representations - 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 [6].

The base class in Java is Object, while in Kotlin it's Any. Kotlin types are divided into nullable and non-nullable. Any is a super-type for all types, but it cannot contain a value of null, the type Any? should be used if a null value is necessary.

III. Operations And Expressions

Operations are the actions that the operands perform when calculating expressions. Table 2 is completed using data from [6], [7], [8] and [9].

TABLE 2. COMPARISON OF OPERATIONS AND EXPRESSIONS IN JAVA AND KOTLIN

Scope resolution	Java	Kotlin
Parenthesis	()	()
Function call	()	()
Array subscript	[]	[]
Access via object	.	., ?.
Post increment	++	++
Post decrement	--	--
Unary plus, minus	+, -	+, -
Logical negation	!	!
Casting operator	(type)	as (type)
Creating object	new	
Multiplicative	*, /, %	*, /, %
Additive	+, -	+, -
Shift left, shift right	<<, >>	shl(bits), shr(bits)
Right with 0 ext	>>>	ushr(bits)
Relational	<, <=, >, >=	<, <=, >, >=
Testing object type	instanceof	is (type)
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 also one of the most important differences between the two languages – Kotlin's explicit support for the so-called nullable types. Putting 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 is necessary to note that the operator used to cast types in Kotlin is as. Similarly to the operator instanceof in Java – in Kotlin the operator is is used to check whether the expression is an instance of a class.

Note that Kotlin does not use the new keyword to create a new object, unlike Java, where its use dedicates memory for the newly created object. In Kotlin, an object is created by calling the constructor just like any ordinary function in the language.

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

There are two types of equality in Kotlin: structural equality (check for equals()) and reference equality (two references point to the same object) [6]. Similar is the Java implementation, where the method equals(), which compares against values in objects and the operator ==, which checks if both objects point to the same memory location, are used to check for equality. 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).

IV. Main Statements In Java And Kotlin

Java and Kotlin 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 JAVA AND KOTLIN

Java	Kotlin
Declaring a variable	
String name;	var name: String
Declaring a variable with initialization	
String name = "Java";	var name = "Kotlin" var name1: String = "Java"
Declaring a named constant	
final int i;	val i: Int

As it can be seen in the example above, when declaring a variable in Java, its type must be specified first and then its name. In Kotlin, the implementation is different - it starts with a keyword, and when declaring a variable with initialization, it is not necessary to specify the type, but it can be explicitly specified. In this case, the compiler parses the initialized expression and uses its type as the variable type.

The keywords that are 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 whose value becomes a constant reference after initialization. The Java counterpart is a variable declared as final.

Executable statements are the assignment statement, conditional branch statements, and loopback statements. Known are three control structures at the level of performance of the statement of the program: sequence, branching (selection, decision), iteration (repetition) [10], which correspond to the types of algorithms – linear, branched or cyclic, of which collate executable statements.

The following table, which shows the syntax of the assignment statement, composite statement and branching statements, is composed by using data from [6], [7], [8] and [9].

TABLE 4. EXAMPLES OF EXECUTABLE STATEMENTS IN JAVA AND KOTLIN

Java	Kotlin
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>;
} else {
<statements>;
}
 	
if (<condition>){
<statements>;
} else if(cond_n){
<statements>;
}else{
<statements>;
}
 	
switch(<intExpression)>{
case <value1>:<statements>;
break;
case <valueN>:<statements>;
break;
default: <statements>;
}
	if(<condition>)
	<statements>
	
	if (<condition>){
	<statements>
	} else {
	<statements>
	}
	
	if (<condition>){
	<statements>
	else if(cond_n){
	<statements>
	}else{
	<statements>
	}
	
	when(<intExpression> {
	<value1> -> <statements>
	<valueN> -> <statements>
	else -> <statements>
	}

Repetition statements	

while(<condition>){
<statements>;
}
 	
for(<init>;<condition>;adjust>)
{
<statements>;
}
 	
do {
<statements>;
}
while<condition>;
	while(<condition>){
	<statements>
	}
	
	do {
	<statements>
	}
	while(<condition>)

Repetition statement based on data structures	

for(<type><identifier>:<array>
{
< statements >
}
	for (item in collection) {
	< statements >
	}

Kotlin supports traditional constructs with the if, while, and do – while statements, but unlike Java if can be used not only as a branching statement but also as an expression. It should be noted here that there is no ternary operator in Kotlin, but its functionality is achieved using the if-else statement.

The branching statement when in Kotlin can be regarded as an expanded version of the statement switch in Java. Left of "- >" is an expression called 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 to the right of the "- >" symbol is satisfied. Conditions are checked sequentially, from top to bottom. If none of the conditions is met, the statement after else is fulfilled.

Unlike Java, Kotlin does not have the traditional for loop, which initializes a variable whose value is updated at every step of the iteration and terminated when a certain value is reached. In Kotlin a for cycle based on data structures is used, equivalent to a foreach loop in Java. This way any data structure that provides an iterator (Range, Array, String, Collection) can be crawled. A range is a set of values defined by two specific endpoint values.

V. SUBROUTINES IN JAVA AND KOTLIN

Java subroutines are called methods and Kotlin subroutines are called functions. Unlike Java, however, in Kotlin it is possible to declare functions outside of classes. Declarations of functions in Kotlin can be combined into namespaces and functions can be declared in a namespace. In addition to the top level in a file, functions can be declared as local, member functions or extension functions.

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

TABLE 5. DIFFERENCES IN IMPLEMENTATION OF SUBROUTINES IN JAVA AND KOTLIN

	Java	Kotlin
Default Argument	–	Default values can provide values to parameters in function definition
Named argument	–	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 to extended with new functionality.
Higher-Order Functions and Lambdas	Lambdas expressions are introduced in the Java 8 and higher-order functions are	Comes as one of the prebuilt features.

	implemented using Callables.	
Tail recursive functions	–	When a recursive function is marked with the tailrec modifier, the compiler optimises out the recursion, without the risk of stack overflow.
Inline Functions	–	Inline function marked, as well as function parameters, should be expanded and generated inline at the call site.

VI. Classes And Objects

Both in Java and in Kotlin objects are created as an instance of a class using constructors. In Java, the declaration of one or more constructors is possible, but in Kotlin 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 constructor helps to incorporate additional logic while initializing the class.

Unlike Java, in Kotlin there are no static class members. For this purpose, it is necessary to use a companion object. Declaration of the object inside the class can be marked with the keyword companion.

In Java the nested classes can be static and non-static. Nested classes declared with the keyword static are static nested classes, while non-static ones are called inner classes. Unlike Java, inner classes in Kotlin cannot access the instance of the outer class, unless specifically requested by adding the modifier inner.

Kotlin data classes are used to define classes that store only properties. These classes do not contain any methods, only properties and, unlike ordinary classes, have no body. In Java, ordinary classes are used for this purpose.

Kotlin, unlike Java, supports integrated delegation (delegation natively by language) as Class Delegation and Delegated Properties. The Class Delegation uses a Delegation pattern which is an alternative to inheritance and makes it possible to implement multiple inheritance.

VII. Conclusion

Our study examined the main structural differences between the two programming languages. It can be concluded that to a large extent Kotlin enhances Java with features such as Smart casts, Extension functions, Delegation, Null-safety, Primary constructors, Companion objects.

With a short and intuitive syntax, Kotlin can increase the productivity of the programmer. For example, the same implementation using a data class only takes one line compared to a few dozen in Java.

Nevertheless, Kotlin does not offer anything fundamentally new. Time will tell whether it will become more popular than Java.

Acknowledgment

The authors would like to thank the Research and Development Sector at the Technical University of Sofia for the financial support.

References

[1] “The 2018 Top Programming Languages”, IEEE Spectrum: Technology, Engineering, and Science News, available online at: https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages, retrieved 3 September 2019.

[2] TIOBE Index | TIOBE - The Software Quality Company, available online at: https://www.tiobe.com/tiobe-index/, retrieved 3 September 2019.

[3] Oracle Documentation, Java™ Platform Overview, available online at: https://docs.oracle.com/javase/7/docs/technotes/guides/, retrieved 4 September 2019.

[4] R. Urma (1 July 2014). "Alternative Languages for the JVM. A look at eight features from eight JVM languages". oracle.com. available online at: https://www.oracle.com/technetwork/articles/java/architect-languages-2266279.html, retrieved 3 September 2019.

[5] C. Haase, "Google I/O 2019: Empowering developers to build the best experiences on Android + Play", available online at: https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html, retrieved 4 September 2019.

[6] Kotlin Language Documentation, available online at: https://kotlinlang.org/docs/kotlin-docs.pdf, retrieved 7 August 2019.

[7] 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

[8] D. Jemerov and S. Isakova, "Kotlin in Action", Manning Publications, Manning Publications; 1 edition (February 19, 2017), ISBN: 978-1617293290

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

[10] S. Bonev, “Programming technology” ("Технология на програмирането"), Siela, Sofia 2000, ISBN: 9546492981