

Introduction to Kotlin

Christian Konersmann, Finn Paul Lippok, Paul Lukas

RWTH Aachen University, Germany

{christian.konersmann,finn.lippok,paul.lukas}@rwth-aachen.de

Proseminar: Advanced Programming Concepts

Supervisor: Jan-Christoph Kassing

March 30, 2025

Abstract

This paper is an introduction to Kotlin, a statically typed, object-oriented programming language designed to be fully interoperable with Java and the Java Virtual Machine (JVM). Kotlin offers a concise syntax, functional programming paradigms, and safety improvements compared to Java. In 2019, Google announced that Kotlin replaced Java as their preferred language for Android development.

1 Introduction

Introduction, motivation, and goals of this paper. This paper assumes that the reader is familiar with the fundamentals of Java.

2 Basic Syntax

This section covers the basic syntax of Kotlin and highlights the differences compared to Java. The goal is to provide a brief overview focused on the most important distinctions.

2.1 Main Method

The main method is the entry point of every Java and Kotlin program. Java enforces object-oriented programming, thus requiring the main method to be declared inside a class. For the main method to be directly executable, the method must be declared as *public* and *static*.

Java main method

```
1 public class Main {  
2     public static void main(String[] args) {  
3         System.out.println("Hello, World!");  
4     }  
5 }
```

Kotlin, on the other hand, does not require methods to be declared inside a class, allowing for a more functional programming style with top-level functions.[3] These top-level functions can be called directly without the need to create an instance of a class, similar to static methods in Java¹ but without class affiliation. Kotlin further reduces boilerplate code by changing the default visibility of everything to public and allowing the main method to be declared without arguments passed as an array.[4, 1] Some further basic syntactical changes include making the semicolon optional and introducing the *fun* keyword for defining functions. These changes lead to a more concise and readable main method and syntax in general.

Kotlin main method

```
1 fun main() {  
2     println("Hello, World!")  
3 }
```

2.2 Type Declaration

In Kotlin, variables are declared using the keyword *val* for immutable variables or *var* for mutable variables, similar to Java's *final* and non-final variables. The type of a variable is declared after the variable name, separated by a colon.

Java data types

```
1 final String name = "JohnDoe";  
2 int age = 42;
```

Kotlin data types

```
1 val name: String = "JohnDoe"  
2 var age: Int = 42
```

2.3 Type Inference

Kotlin supports type inference, allowing the compiler to infer the type of a variable based on its initializer.

```
1 val name = "JohnDoe" // type is inferred as String  
2 var age = 42 // type is inferred as Int
```

2.4 Method Declaration

Similar to java. void = Unit which is optional.

Java method declaration

```
1 public int add(int a, int b) {  
2     return a + b;  
3 }
```

¹When compiling Kotlin to Java bytecode, top-level functions are compiled to static methods in a class named after the file name.

Kotlin method declaration

```
1 fun add(a: Int, b: Int): Int {  
2     return a + b  
3 }
```

2.5 Everything is an Object

In Kotlin, there are no primitive types.² All types are objects and inherit from the `Any` class.[2] This approach creates a more consistent object-oriented programming model and eliminates the need for wrapper classes.

Java Integer Wrapper

```
1 Integer.valueOf(42).hashCode();
```

Kotlin direct usage of Int

```
1 42.hashCode()
```

Furthermore, functions in Kotlin are also objects. This enables higher-order functions and functional programming paradigms, meaning that functions can be passed as arguments, returned from other functions, and assigned to variables.

3 New Language Constructs

This section focuses on the most important new language constructs that are not present in Java. This section will illustrate Kotlin's advantages using a list of salespersons as an example and comparing it to Java. The example should represent a linked list of salespersons containing multiple attributes.

3.1 Classes

In both Java and Kotlin, classes are declared using the `class` keyword and can contain attributes, methods, and constructors. In this example, we declare a class to hold information about a salesperson.

Java Class Declaration

```
1 public class SalesPerson {  
2     private final String name;  
3     private final int commissionRate;  
4     private double salesVolume;  
5  
6     public SalesPerson(String name, int commissionRate, double  
7         transferAmount) {  
8         this.name = name;  
9         this.commissionRate = commissionRate;  
10        this.salesVolume = transferAmount;  
11    }  
}
```

²Certain types may be optimized to use primitives during runtime for performance reasons.

Kotlin improves upon Java by allowing the constructor to be declared directly within the class definition. As a reminder, public is the default visibility in Kotlin. In addition, classes are also final by default, meaning they cannot be inherited from unless explicitly declared as open. Furthermore, Kotlin allows for the declaration of attributes and their visibility directly within the constructor by adding the *val* or *var* keyword and the private keyword, resulting in a syntax very similar to Java records.

Kotlin Class Declaration

```
1 class SalesPerson(val name: String, private val commissionRate:
   Int, transferAmount: Double = 0.0) {
2     var salesVolume: Double = transferAmount
3 }
```

In this example, *name* and *commissionRate* become properties of the class, while *transferAmount* is a constructor parameter used to initialize the property *salesVolume*. It is still possible to declare attributes outside of the constructor, as demonstrated by *salesVolume*. In addition, Kotlin allows default parameter values in constructors and functions, a feature that would otherwise require method overloading in Java.

3.2 Properties

Explain properties, getters, and setters. Build off the previous example and show how to use properties in Kotlin (visibility modifiers, custom getters, and setters).

Kotlin Properties

```
1 var salesVolume: Double = transferAmount
2     private set(value) {
3         if (value < 0)
4             throw IllegalArgumentException("Sales volume must be
               positive")
5         field = value
6     }
7
8 val commission: Double
9     get() = salesVolume * commissionRate
```

3.3 String Interpolation

A very short introduction to string interpolation.

String Interpolation

```
1 fun printSalesPerson() {
2     println("Name: $name, Sales in USD: ${salesVolume * 1.2} \$")
3 }
```

3.4 Null Safety

Whenever a method or an attribute is called on a null reference in Java, a *NullPointerException* (NPE) is thrown. The concept behind Null Safety aims to reduce the occurrence of such NPEs. This is achieved through the advanced type system of Kotlin, which distinguishes between nullable and non-nullable types. This guarantees that variables of a non-nullable type can never be null. Unlike Java, this is enforced by the compiler at

compile-time, therefore reducing possible sources of NPEs and enhancing the readability and robustness of the code. At runtime, both types are treated the same.

By default, all types in Kotlin are non-nullable, meaning variables cannot hold a null value unless explicitly specified. To allow nullability, a question mark is appended to the type declaration.³

```
1 var a: String = "a_is_non-nullable"
2 var b: String? = "b_is_nullable"
```

3.4.1 Null Safety Operators

When working with nullable types, you cannot directly access properties or methods because the value could be null, potentially causing an NPE. Whenever a nullable type is used, the value must be checked in some way to prevent the program from encountering an NPE. To avoid excessive use of if statements, Kotlin provides the safe call operator as a shortcut.

The *safe call operator* consists of the characters `?.` and is used when accessing a property or method of a nullable object. If the object is null, the operator returns null without evaluating the rest of the expression. Otherwise, the expression is evaluated as usual. Practically, this operator extends the already familiar dot notation for attributes and methods of objects. In principle, the safe call operator can also be seen as a shorthand for an if statement. By using the safe call operator, the code becomes much more readable and concise. With the reduced complexity, it is also less error-prone.

Using the safe call operator in comparison to an if statement

```
1 var couldBeNull: String? = null
2 println(if (couldBeNull == null) null else couldBeNull.length)
3 println(couldBeNull?.length) // Safe call operator
```

We can use multiple safe call operators and chain them together. The compiler evaluates the expression from left to right, checking each operator sequentially. If any value is null, the entire expression evaluates to null. Furthermore, the operator can also be used on the left side of an assignment. If the safe call operator evaluates to null, the assignment will be skipped. Otherwise, the value will be assigned as usual.

```
1 var age: Int? = rwth?.ceo?.age // chained safe call operators
2 rwth?.ceo?.age = 20 // assignment with chained operator
```

In Java, there is no safe call operator or anything comparable. To prevent NPE in Java, you have to explicitly check with an if-statement for the value to not be null. This is very inconvenient, hard to read, and prone to errors.

Prevent NPE in Java

```
1 String couldBeNull = null;
2 if (couldBeNull == null) System.out.println("null");
3 else System.out.println(couldBeNull.length());
```

The *Elvis operator* is an extension of the safe call operator. The Elvis operator is mostly used in combination with the safe call operator. If the left side is null, the value

³This applies to both mutable and immutable variables.

will be set to a default value. Otherwise, the expression to the right-hand side will just be evaluated... TODO

3.4.2 Smart Casts

Smart casts are another way to simplify the usage of nullable variables.

4 Interoperability

5 Multiplatform development

5.1 Expected and actual declarations

5.2 Hierarchical project structure

6 Android

This sections concentrates on the benefits Kotlin has in the Android enviornment and gives examples based on the salesperson example from New Language Constructs along the way.

6.1 Android KTX

6.2 Jetpack Compose

TODO(Kotlin based Ui Tool Kit JC automaticly updates Ui hierarchy functions with @Composable /composables)

6.3 Coroutines

Now imagine we want to create an app where users can see the sales volume of a certain salesperson live. To achieve this, we would need to check the sales volume every second, which would freeze our app every second until the data of the salesperson is successfully downloaded. So, the smartest solution would be to use multithreading for this process.

What are threads? -> explanation ... todo best case with graphic

Java Background Threads

```
1 new Thread(new Runnable() { //opens a new thread
2     public void run() {
3         double sales = getSalesVolume();
4         runOnUiThread(new Runnable() { //switches to main thread
5             public void run() {
6                 textView.setText(sales.toString());
7             }
8         });
9     }
10 }).start();
```

In Kotlin, threads are called coroutines, and they are not only easy to read, as you will see below, but are also very lightweight, which means we can run far more Kotlin coroutines than Java threads before running out of memory or losing too much time. So, we could check thousands of sales personnel at once without running out of memory.

Kotlin Coroutines

```
1 GlobalScope.launch { //opens a new thread
2     val sales = getSalesVolume()
3     withContext(Dispatchers.Main) { //switches to main thread
4         textView.text = sales
5     }
6 }
```

6.4 Extensions

7 TODO

7.1 General

- Add citations
- Fix formatting (especially indentation in the code snippets)

7.2 Introduction

Android development, improvements over, and interoperability with Java. Introduce an example to show differences/translation between Java and Kotlin.

7.3 Basic Syntax

- Methods
- Example for Top-Level Functions
- Explain the absence of static methods (out of scope for introduction?) (use `@JvmStatic` annotation for interoperability)

7.4 Interoperability

- Explain the interoperability between Java and Kotlin (e.g. Using Java libraries in Kotlin)
- Use of annotations (e.g. `@JvmStatic`, `@JvmField`, `@JvmName`, `@JvmOverloads`) (out of scope for introduction?)
- Compile to other languages (e.g. JavaScript, Native) (out of scope for introduction?)

7.5 New Features

- Change Null Safety example to build off the previous example
- Properties (Getters, Setters)
- Extension functions (not as important)
- This expression (interesting, also not too long)
- Destructuring declarations
- Infix notation for functions
- *if* and *when* as expressions (not as important, only if it fits)

7.6 Multiplatform development

7.7 Android

- Discuss Kotlin's advantages for Android development

References

- [1] Kotlin programming language. *Basic Syntax. Program-entry-point*. Kotlin documentation. JetBrains. Nov. 6, 2024. URL: <https://kotlinlang.org/docs/basic-syntax.html#program-entry-point> (visited on 03/24/2025).
- [2] Kotlin programming language. *Basic Types*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/basic-types.html> (visited on 03/29/2025).
- [3] Kotlin programming language. *Functions. Function-scope*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/functions.html#function-scope> (visited on 03/24/2025).
- [4] Kotlin programming language. *Visibility Modifiers*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/visibility-modifiers.html> (visited on 03/24/2025).