

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

April 1, 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:
2      Int, transferAmount: Double = 0.0) {
3      var salesVolume: Double = transferAmount
4  }
```

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
5                  positive")
6          field = value
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 Extension functions

3.5 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"
```

3.5.1 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
```

The *Elvis operator* (`?:`) is an enhanced version of the safe call operator, offering a more concise way to handle null values. If the expression on the left side of the Elvis operator evaluates to null, instead of returning null like the safe call operator, it returns a default value specified on the right side. As a result, the Elvis operator is commonly used alongside the safe call operator. In essence, both operators serve as simplified alternatives to if statements. This shorthand improves code readability and maintainability.

Using the Elvis operator in comparison to an if statement

```
1 var couldBeNull: String? = null
2 println(if (couldBeNull == null) 0 else couldBeNull.length)
```

³This applies to both mutable and immutable variables.

```
3 println(couldBeNull?.length ?: 0) // Elvis operator
```

Java does not have a safe call operator, an Elvis operator, or any equivalent feature. 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());
```

Both the safe call operator and the Elvis operator are treated by the compiler as the if statements mentioned in the examples above. It simply makes the code significantly shorter and easier to read.

3.5.2 Safe casts

Safe casts are another way to handle nullable objects. But in order to understand the Safe cast, we have to look at how Kotlin handles type casts in general. The principle behind casting is the same as in java, only the syntax is different. Kotlin uses the `as` keyword behind the expression followed by the new type to cast one type into another. In java the new type had to be written in round brackets before the expression.

Casting in Kotlin

```
1 var a: Any = "Replace this example"
2 var b: String = a as String
```

Safe cast is used to prevent a `ClassCastException` when a given object does not match the target type. The safe cast operator extends the standard cast keyword by adding a question mark and is applied in the same manner as a regular cast. If the object is not of the target type, instead of throwing an exception, the expression evaluates to null. This significantly simplifies casting, eliminating the need to catch potential exceptions or perform type checks⁴ beforehand. The functionality of the operator can also be replicated using if statements, further demonstrating its benefits for code readability and maintainability.

Usage of the safe cast operator in comparison to an if statement

```
1 var str: Any? = "Also replace *this* example"
2 var a: Int? = str as? Int // evaluates to null
3 var b: Int? = if (str is Int) str else null // no need for the
               'as Int' here due to smart casting
```

But the safe cast operator is like the other two null safety operators very useful at handling nullable objects. If the argument of the safe cast is null, instead of throwing a NPE the expression evaluates to null as well. Therefore the code is less prone to errors. If you want to enhance null safety, you can combine the safe cast operator with the Elvis operator to provide a fallback value when the safe cast operator returns null due to a type mismatch or a null reference.

⁴Type checks in Kotlin are performed using the `is` and `is!` keywords, which function similarly to the `instanceof` keyword in Java.

Usage of the safe cast operator on a nullable value

```
1 var str:Any? = null
2 var a:Int? = str as? Int // evaluates to null
```

As mentioned above there is nothing like the safe cast operator in Java. If you wanted to achieve the same result, you either had to catch the `ClassCastException` or had to check for nullability before casting. This once again demonstrates how Kotlin's concise and well-designed syntax significantly simplifies programming compared to Java.

Functionality of safe call operator in java

```
1 Object obj = null;
2 String str = null;
3 if (obj != null && obj instanceof String s) str = s;
```

3.5.3 Not-null assertion

The *not-null assertion operator* consists of two exclamation marks (!!). It is used to convert nullable types to non-nullable types by instructing the compiler to treat the value as non-null. However, if the value is actually null, a `NullPointerException` (NPE) will be thrown. This operator contradicts the concept of null safety and should only be used when the programmer is certain that the value cannot be null, but the compiler is unable to guarantee it.

Usage of the not-null assertion

```
1 var couldBeNull: String? = null
2 var b: String = couldBeNull!!
```

3.5.4 Nullable receiver

We have already covered extension functions in the chapter on Classes. As a brief reminder, extension functions are external additions to a class that introduce new methods, which can be called on an instance of the class using dot notation.

Since extension functions are not actually part of the class itself but merely an extension that can be called using dot notation, it is possible for the object to be null while still being able to call the extension method. To achieve this, the function must have a so-called *nullable receiver* type, which is indicated by a question mark after the class the extension function is defined for. As a result, the method remains accessible even if the object is null. This allows values of a nullable type to be accessed without checking for null beforehand, as the null case is handled within the method itself. The following example demonstrates how to define and properly use an extension function with a nullable receiver type.

Usage of an extension function

```
1 // define the extension function
2 fun SalesPeron?.print() {
3     if (this == null) return println("This person dose not exist.")
4     return println("$name: $salesVolume sold")
5 }
6 // use the extension function
7 var sales: SalesPeron? = null
8 sales.print() // This person dose not exist.
```

```

9     sales = SalesPeron("Carl", 1200)
10    sales.print() // Carl: 0.0 sold

```

3.5.5 Collections of nullable types

When working with Collections of nullable types, it is often very inconvenient to always handle the possible null cases. To avoid this, there are two alternative options which make things a lot easier. In the following example we have a List with Strings, that could be null. To not have to care about null values, there is a function called `.filterNotNull()` which removes all of the null values of the list and returns a List with the corresponding not-nullable type.

```

1     val nullList: List<SalesPeron?> = listOf(SalesPeron("Carl",
2         2300), null)
3     val list: List<SalesPeron> = nullList.filterNotNull()
4     println(list) // prints SalesPeron@c4437c4

```

Another useful option is the *let function*, one of Kotlin's so-called scope functions, often used when working with lists of nullable types. This function takes a lambda expression and returns its result. Within the lambda, the object can be accessed using the `s` keyword. Essentially, `let` is an extension function available for every type in Kotlin, executing a given code block when invoked. If the `let` function is used with the safe call operator and the object is null, the safe call operator prevents further evaluation, ensuring that the lambda's code block is not executed.

```

1     val nullList: List<SalesPeron?> =
2         listOf(SalesPeron("Carl", 2300), null)
3     for (pers: SalesPeron? in nullList) {
4         pers?.let { println(it.name) }
5     }

```

4 Interoperability

In this chapter, the focus will be on interoperability between Java and Kotlin. In this context, interoperability means that the two languages are compatible with each other. Kotlin was designed to allow seamless integration of Java code in Kotlin and vice versa. This is possible because both languages compile to Java bytecode. This concept is especially useful when working with libraries. There are already countless libraries written in Java that can now be used in Kotlin, eliminating the need to rewrite a library with the same functionality specifically for Kotlin. This applies to both the official Java standard libraries and more specialized external libraries. Additionally, interoperability makes it much easier to migrate existing Java projects to Kotlin, as they do not need to be completely rewritten. This once again shows that Kotlin is a well-thought-out language designed to serve as an improvement over Java.

4.1 Call Java in Kotlin

One of the key aspects of Kotlin is its interoperability with Java. This means that Kotlin was specifically designed to support the use and execution of any Java code within a Kotlin project. To illustrate how Java code can be accessed, the following example features a Salesman class that stores basic information using getters and setters.

Example java class

```
1 public class Salesman {
2     private final String name;
3     private int salary;
4
5     public Salesman(String name, String title, int salary) {
6         this.name = name;
7         this.salary = salary;
8     }
9
10    public String getName() { return name; }
11
12    public int getSalary() { return salary; }
13
14    public void setSalary(int salary) { this.salary = salary; }
15 }
```

If we want to access this class from Kotlin and create an instance of it, we can use the familiar Kotlin syntax to instantiate the object and access its properties. There is no syntactical difference between calling or creating a Java class and a Kotlin class. Since getter and setter methods are unnecessary in Kotlin, they are automatically converted if they follow Java conventions for getter and setter methods. This allows them to be accessed using Kotlin's property syntax. The resulting attributes are called synthetic properties. If the getters and setters do not follow Java conventions, they can still be accessed as regular methods.

Access the Salesman class in Kotlin

```
1 var carl = Salesman("carl_mueller", 4500)
2 println(carl.name) // prints 'carl mueller'
3 carl.salary = 4600 // sets salary to 4600
4 carl.setSalary(4600) // alternivly to the above
```

Kotlin detects that the name field in the Java class is final and has a getter but no setter. As a result, the compiler throws an error if an attempt is made to modify its value, ensuring that the getters and setters behave the same way as in Kotlin. If the field had only a setter, the method would not be converted into a synthetic property, as Kotlin does not support set-only properties.

There are a few keywords, such as *in* or *is*, that do not exist in Java; therefore, they are valid names for variables or similar identifiers. If there is Java code using those keywords, it is still possible to interact with this code through the backtick (`) character. In the following example, there is a method of a Java class called *in* we want to access:

```
1 var salesman = Salesman("freddy", 1300)
2 salesman.`in`(list)
```

- null safety with Java-Classes
- annotations
- mapped Types
- access static members
- arrays

4.2 Call Kotlin in Java

It is possible to call Kotlin-Files in Java-code as well.

- properties
- static fields

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 Conclusion

- concise syntax
- improved classes
- new features like null safety
- interoperability with java and JS
- Multiplatform development -> especially Android
- Also interesting: functional programming in kotlin
- Also interesting: ...

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).

8 TODO

8.1 General

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

8.2 Introduction

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

8.3 Basic Syntax

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

8.4 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?)

8.5 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)

8.6 Multiplatform development

8.7 Android

- Discuss Kotlin's advantages for Android development