# Introduction to Kotlin

Christian Konersmann, Finn Paul Lippok, Paul Lukas

RWTH Aachen University, Germany
{christian.konersmann,finn.lippok,paul.lukas}@rwth-aachen.de

March 24, 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. This paper was written as part of the *Proseminar: Advanced Programming Concepts*.

## 2 Basic Syntax

This section will cover the basic syntax of Kotlin and highlight the changes compared to Java. The goal of this section is to provide a brief overview, focusing on the most important differences.

### 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 static and public.

Java main method

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

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. These top-level functions can be called directly without the need to create an instance of a class, similar to static methods in Java[1] 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. 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

```kotlin
fun main() {
  println("Hello, World!")
}
```

## 2.2 Type Declaration

In Kotlin, a variable declaration starts with 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

```java
    final String name = "John Doe";
    int age = 42;
```

Kotlin data types

```kotlin
    val name: String = "John Doe"
    var age: Int = 42
```

## 2.3 Type Inference

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

```kotlin
    val name = "John Doe" // type is inferred as String
    var age = 42 // type is inferred as Int
```

## 2.4 Method Declaration

Similar to java. void = Unit which is optional.

Java method declaration

```java
    public int add(int a, int b) {
      return a + b;
    }
```

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

Kotlin method declaration

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

### 2.5 Everything is an Object

In Kotlin, everything is an object, including primitive types and functions.

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

Java Class Declaration

```
public class SalesPerson {
  private final String name;
  private final int commissionRate;
  private double salesVolume;

  public SalesPerson(String name, int commissionRate, double
      transferAmount) {
      this.name = name;
      this.commissionRate = commissionRate;
      this.salesVolume = transferAmount;
    }
  }
```

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

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

In this example, both *name* and *commissionRate* are attributes, while *transferAmount* is a normal constructor parameter. These constructor parameters can be used to initialize attributes of the class. It is still possible to declare attributes outside of the constructor, like

*salesVolume* in this example. In addition, Kotlin allows for default values for parameters in constructors and functions, which would require overloading in Java.

## 3.2 Properties

## 3.3 String Interpolation

## 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.[2]

```
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.

[2]This applies to both mutable and immutable variables.

```
var age: Int? = rwth?.ceo?.age // chained safe call operators
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

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

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

```
var couldBeNull: String? = null
println(if (couldBeNull == null) 0 else couldBeNull.length)
println(couldBeNull?.length ?: 0)   // Elvis operator
```

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 Not-null assertion

### 3.5.3 Let function

### 3.5.4 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 diffrent. 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 brakets before the expression.

Casting in Kotlin

```
var a: Any = "Replace␣this␣example"
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 checks 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

```
var a: String = "Also replace *this* example"
var b:Int? = a as? Int // evaluates to null
```

# 4 Interoperability

# 5 TODO

## 5.1 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?)

## 5.2 New Features

- Null safety
- Properties (Getters, Setters)
- Extension functions (not as important)
- This exrpession
- Destructuring declarations
- *if* and *when* as expressions (not as important, only if it fits)


# Acknowledgements

We would like to thank our instructor.