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
Organiser: Prof. Dr. Jürgen Giesl
Supervisor: Jan-Christoph Kassing

April 7, 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

```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.[14] 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.[35, 5] 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, 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

```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 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

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

2.5 Everything is an Object

In Kotlin, there are no primitive types.[2] All types are objects and inherit from the `Any` class.[6] This approach creates a more consistent object-oriented programming model and eliminates the need for wrapper classes.

Java Integer Wrapper

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

Kotlin direct usage of Int

```kotlin
  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 Salesperson class as an example and comparing it to Java. The example represents a salesperson 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

```java
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;
    }
  }
```

[2] Certain 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

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

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

Properties allow for the concise declaration of getters and setters, including their visibility, directly alongside the corresponding attribute. In contrast to Java, when accessing a property in Kotlin using dot notation, the getter or setter method is called internally, allowing for a more universal syntax. If only the visibility needs to be restricted, the property can be declared as shown below:

Private setter

```
var salesVolume: Double = transferAmount
  private set
```

For more complex logic, custom getter and setter can be defined directly within the property declaration. The *field* keyword refers to the attribute itself.

Kotlin Properties

```
var salesVolume: Double = transferAmount
  private set(value) {
    if (value < 0)
      throw IllegalArgumentException("Sales volume must be
          positive")
    field = value
  }

val commission: Double
  get() = salesVolume * commissionRate
```

3.3 String Interpolation

A very short introduction to string interpolation.

String Interpolation

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

```
}
```

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 [24]. This is achieved through the advanced type system of Kotlin, which distinguishes between nullable and non-nullable types [29]. 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]

```
    var a: String = "a is non-nullable"
    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 [10], 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 [31]. 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

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

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

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

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 [12]. 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
```

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

```
String couldBeNull = null;
if (couldBeNull == null) System.out.println("null");
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 diffrent. Kotlin uses the `as` keyword behind the expression followed by the new type to cast one type into another [3]. 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 [32]. 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 [4] 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 comparisopn to an if statement

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

[4]Type checks in Kotlin are performed using the is and !is keywords [16], which function similarly to the instanceof keyword in Java.

```
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 usefull 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.

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 [22]. 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 [28]. 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

```
// define the extension function
fun SalesPeron?.print() {
    if (this == null) return println("This person dose not
        exist.")
    return println("$name: $salesVolume sold")
}
// use the extension function
var sales: SalesPeron? = null
sales.print() // This person dose not exist.
sales = SalesPeron("Carl", 1200)
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 possibble null cases. To avoid this, there are two alternativ options witch makes things a lot easyer. 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()* wich removes all of the null values of the list and returns a List with the corresponding not-nullable type [11].

```
val nullList: List<SalesPeron?> = listOf(SalesPeron("Carl",
    2300), null)
val list: List<SalesPeron> = nullList.filterNotNull()
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 [20]. 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.

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

4 Interoperability

This chapter focuses on interoperability between Java and Kotlin. In this context, interoperability refers to the seamless compatibility between the two languages. Kotlin was designed to integrate smoothly with Java code and vice versa, making it easy to use both within the same project.

4.1 Call Java in Kotlin

Everything written in Java is accessible in Kotlin, but interoperability is especially useful when working with Java 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.

4.1.1 Create and acces objects

Kotlin was specifically designed to support the use and execution of any Java code within a Kotlin project [8]. 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

```java
public class Salesman {
  private final String name;
  private int salary;

  public Salesman(String name, String title, int salary) {
    this.name = name;
    this.salary = salary;
  }

  public String getName() { return name; }

  public int getSalary() { return salary; }

  public void setSalary(int salary) { this.salary = salary; }
}
```

If we want to access this class from Kotlin and create an instance of it, we can use the familiar Kotlin syntax [8] 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 [4]. This allows them to be accessed using Kotlin's property syntax. The resulting attributes are called synthetic properties [17]. If the getters and setters do not follow Java conventions, they can still be accessed as regular methods.

Access the Salesman class in Kotlin

```kotlin
var carl = Salesman("carl␣mueller", 4500)
println(carl.name) // prints 'carl mueller'
carl.salary = 4600 // sets salary to 4600
carl.setSalary(4600) // alterniuly 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 [17].

4.1.2 Mapped types

By default, when objects of a Java class are used in Kotlin, they are loaded as Java objects. However, some Java types have a corresponding Kotlin counterpart, and the Java object is automatically replaced with the equivalent Kotlin type [21]. For example, 'java.lang.Integer' is converted to 'kotlin.Int?' because Java wrapper types can be nullable. This applies to all Java wrapper classes and some important types, such as 'java.lang.Object', which is mapped to 'kotlin.Any!'[5]. Collections like Lists, Maps and Arrays are also converted. Additionally, all Java primitive types are mapped to their non-nullable Kotlin counterparts, as primitive types cannot be null in Java. For instance, the Java 'int' is converted to 'kotlin.Int'. For a complete list of all mapped types, refer to the official documentation.

4.1.3 Null safety with Java

Since Java does not distinguish between nullable and non-nullable types, any object returned from Java code can be null. This contradicts Kotlin's strict null safety concept and would make working with Java objects impractical. To address this, Kotlin introduces *platform types* for objects created through Java code. If a Java type does not have a direct Kotlin equivalent, as is the case for most Java types, the compiler assigns it a platform type, which is non-denotable [26]. This means we cannot explicitly declare or write this type as we do with nullable types using a question mark[6]. With platform types, Kotlin relaxes its strict null safety rules, making their handling similar to Java. However, this increases the risk of NullPointerExceptions. To see how we can use this in practice, we have extended the previously introduced Java Salesman class with the following method:

```
1     public static List<Salesman> createList() {
2       List<Salesman>  list = new ArrayList<>();
3       list.add(null);
4       list.add(new Salesman("Carl", 4200));
5       return list;
6     }
```

If we access this method through Kotlin, we get the List containing the two Elements created in Java. Since both Objects are created in Java and could be null, they are assigned the platform type, thus the developer can decide if the variable should be nullable or non-nullable.

```
1     val list = Salesman.createList()
2     println(list::class.qualifiedName)
3     var item: Salesman = list[0]
4     var nullableItem: Salesman? = list[1]
5     println(item.name) // allowed but would throw NPE
```

If we set the type to non-nullable but the object is actually null, attempting to access its members will result in a NullPointerException, as shown above. Therefore, it is always safer to use nullable types.

Some Java compilers use annotations [27] to specify whether a value is nullable or non-nullable, such as JetBrains' @Nullable or @NotNull annotation [2]. If these annotations are present in the Java code, the compiler assigns the corresponding nullable or non-nullable

[5]The exclamation mark indicates that this is a platform type. More on this in the next chapter.

[6]When the compiler needs to report a type-related error, it uses an exclamation mark to indicate the platform type [23].

Kotlin type to the variable instead of a platform type. If we had a method returning a simple String with a @NotNull annotation in our Salesman class, the variable would actually assigned the non-nullable type instead of the platform type:

```
    public static @NotNull String getString() {
      return "Not null";
    }
```

```
    val str: String = Salesman.getString() // non-nullable type
```

4.1.4 Java arrays in Kotlin

In Java, arrays of primitive types can be used to achieve better performance, as they avoid the overhead associated with objects. Kotlin prohibits the direct use of primitive arrays but provides specialized classes for each primitive type instead [19]. The compiler optimizes the code and uses primitive arrays whenever possible. For example, Java's 'int[]' corresponds to Kotlin's 'IntArray'. These classes compile down to actual primitive arrays to minimize object overhead.

Let's assume we have a function in Java that requires a primitive array:

```
    public static void takeArray(int[] array) { ... }
```

To call this function from Kotlin without unnecessary boxing, we should use intArrayOf() instead of arrayOf(). This ensures that the array compiles down to Java's 'int[]', avoiding the overhead of boxed Integer objects. Even in for loops, the Kotlin compiler optimizes iteration over primitive arrays, ensuring that no iterator is created. This results in significant performance improvements compared to iterating over an Array<Int>, which would involve additional function calls and object overhead.

```
    var array: IntArray = intArrayOf(1, 2, 3)
    takeArray(array) // passes int[] to Java function
    for (i in array.indices) // no iterator created
        println(array[i]) // no calls to Array's get() or set()
```

In Java, arrays are covariant, meaning an array of a subtype can be assigned to an array of its superclass. This is allowed at compile time, but Java enforces type safety at runtime. If an instance of a type that differs from the array's original type is assigned to it, an ArrayStoreException will be thrown. This happens because mixing different types in the array would break type safety. To counteract this problem, Kotlin simply dose not allow this, thus there arrays are invariant [19]. However there is an exception when you need to parse an array to Java, it is allowed for platform types, because Java treates arrays as covariant. If we had a method, that requires a Object array, we could parse a string array of platform type to it.

```
    public static void takeArray(Object[] array) { ... }
```

```
    var array: Array<String> = arrayOf("string", "array")
    takeArray(array) // array is treated as platform type
```

4.1.5 Interference between Kotlin keywords and Java identifiers

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 it using the backtick (‘) character [13]. In the following example, there is a method of a Java class called *in* we want to access:

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

4.2 Call Kotlin in Java

Just as Kotlin can create instances of Java classes, Java can also create and use instances of Kotlin classes [9].

4.2.1 Kotlin properties in Java

If you want to access a Kotlin class from Java, you need to use Java syntax, so it is not possible to access Kotlin properties directly as you would in Kotlin. To bridge this gap, Kotlin properties are compiled into a private field, along with corresponding getter and setter methods [18]. However if the Kotlin property is final, no setter method will be created. For example, consider a simple property in the SalesPerson class:

```
var name: String
```

This will compile to the following components in Java:

```
private String name;
public String getName() { return name; }
public void setName(String name) { this.name = name; }
```

If the getter or setter of a Kotlin property is declared with restricted visibility, such as private or protected, the Kotlin compiler will respect this when generating the corresponding Java methods, thereby limiting their accessibility.

4.2.2 Null safety

If a public Kotlin function is called from Java, it is possible to pass any object to that function, even if the type is non-nullable in Kotlin. To retain null safety, Kotlin generates checks for those functions and throws a NullPointerException if the value is indeed null [25].

4.2.3 Package-level function

Package-level functions are functions in a class, wich are defined outside of classes, thus are not dependent on an object. Thouse functions including extension functions will be converted to a static methods inside of a new Java class [30] named by the Kotlin file the function oriented from, because in Java methods outside an class are not prohibited. For example, we have a Kotlin file with the name ’SalesPerson.kt’ inside of the package ’org.company’ with the follwing package-level method:

```
// SalesPerson.kt
package org.company

fun createDefault(name: String) { ... }

class SalesPerson(var name: String, var salary: Int) { ... }
```

The class will be accessable just like normal, but the createDefault function is in a seperate class called 'SalesPersonKt.class':

```
// create instance as expected
new org.company.SalesPerson("Carl", 4200);
// createDefault in other class
org.company.SalesPersonKt.createDefault("Carl");
```

We can choose a name for the generated class with the '@file:JvmName("Example")' annotation inside the Kotlin file:

```
// SalesPerson.kt
@file:JvmName("Example")
package org.company
...
```

```
// createDefault in Example
org.company.Example.createDefault("Carl");
```

If multiple classes use the same name, the compiler would normally throw an error. However, by adding the @file:JvmMultifileClass annotation to all of them, all package-level functions with the same class name are combined into a single generated class [30].

4.2.4 Instance fields

In Java, it is possible to access public attributes without using getter and setter methods. This kind of direct access is prohibited in Kotlin. However, if the class is accessed from Java, we can add the @JvmField annotation before our property to make it accessible in Java through dot notation [15]. The field will have the same visibility as the property in Kotlin [7]. This example demonstrates how to use the annotation.

```
class SalesPerson (@JvmField var name:String) {}
```

```
public void example() {
  SalesPerson person = new SalesPerson("Carl");
  System.out.println(person.name); // prints 'Carl'
}
```

Functions in companion or named objects can also be marked as static for Java interoperability using the same annotation [34]; however, this topic is beyond the scope of this paper.

4.3 Interoperability with JavaScript

Besides interoperability with Java, Kotlin also supports interoperability with JavaScript [33], a scripting language that runs in both the browser and on Node.js servers. To achieve this, you need to create a Kotlin/JS project and compile it with Gradle, a build automation tool commonly used in the Kotlin and Java ecosystems [1], into .js files, which can then be used like regular JavaScript files. Unlike Kotlin/Java interoperability, a Kotlin/JS project requires more setup due to the Gradle build system and the specialized Kotlin-to-JavaScript compiler. However, the Kotlin documentation [33] provides a step-by-step setup guide. With this setup, it is possible to use JavaScript libraries or the DOM API [7] for web development using the familiar Kotlin syntax, thus making it a valid alternative

[7] This does not apply to private properties.

to plain JavaScript. Furthermore, Gradle provides features that improve the workflow and simplify the development process.

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 enviorment 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? -> explenation ... todo best case with graphic

Java Background Threads

```
new Thread(new Runnable() { //opens a new thread
    public void run() {
        double sales = getSalesVolume();
        runOnUiThread(new Runnable() { //switches to main thread
            public void run() {
                textView.setText(sales.toString());
            }
        });
    }
}).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

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

6.4 Extensions

7 Conclusion

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

References

[1] Gradle Inc. *What is gradle?* Gradle documentation. Gradle Inc. 2024. URL: https://docs.gradle.org/current/userguide/userguide.html (visited on 04/05/2025).

[2] JetBrains. *JetBrains Nullability Annotations*. JetBrains documentation. JetBrains. Jan. 6, 2025. URL: https://www.jetbrains.com/idea/help/nullable-and-notnull-annotations.html (visited on 04/03/2025).

[3] Kotlin programming language. *"Unsafe" cast operator*. Kotlin documentation. JetBrains. Nov. 6, 2024. URL: https://kotlinlang.org/docs/typecasts.html#unsafe-cast-operator (visited on 04/05/2025).

[4] Kotlin programming language. *Access getters and setters in Java*. JetBrains documentation. JetBrains. Jan. 6, 2025. URL: https://kotlinlang.org/docs/java-interop.html#getters-and-setters (visited on 04/21/2025).

[5] 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).

[6] Kotlin programming language. *Basic Types*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/basic-types.html (visited on 03/29/2025).

[7] Kotlin programming language. *Browser and DOM API*. Kotlin documentation. JetBrains. Feb. 15, 2021. URL: https://kotlinlang.org/docs/browser-api-dom.html (visited on 04/05/2025).

[8] Kotlin programming language. *Calling Java from Kotlin*. JetBrains documentation. JetBrains. Jan. 21, 2025. URL: https://kotlinlang.org/docs/java-interop.html (visited on 04/03/2025).

[9] Kotlin programming language. *Calling Kotlin from Java*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/java-interop.html#java-arrays (visited on 04/05/2025).

[10] Kotlin programming language. *Check for null with the if conditional*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/null-safety.html#check-for-null-with-the-if-conditional (visited on 04/05/2025).

[11] Kotlin programming language. *Collections of a nullable type*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/null-safety.html#collections-of-a-nullable-type (visited on 04/05/2025).

[12] Kotlin programming language. *Elvis operator*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/null-safety.html#elvis-operator (visited on 04/05/2025).

[13] Kotlin programming language. *Escaping for Java identifiers that are keywords in Kotlin*. JetBrains documentation. JetBrains. Jan. 21, 2025. URL: https://kotlinlang.org/docs/java-interop.html#escaping-for-java-identifiers-that-are-keywords-in-kotlin (visited on 04/03/2025).

[14] 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).

[15] Kotlin programming language. *Instance fields*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/java-to-kotlin-interop.html#instance-fields (visited on 04/05/2025).

[16] Kotlin programming language. *is and !is operators*. Kotlin documentation. JetBrains. Nov. 6, 2024. URL: https://kotlinlang.org/docs/typecasts.html#is-and-is-operators (visited on 04/05/2025).

[17] Kotlin programming language. *Java synthetic property references*. JetBrains documentation. JetBrains. Jan. 21, 2025. URL: https://kotlinlang.org/docs/java-interop.html#java-synthetic-property-references (visited on 04/03/2025).

[18] Kotlin programming language. *Kotlin properties in Java*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/java-to-kotlin-interop.html#properties (visited on 04/05/2025).

[19] Kotlin programming language. *Kotlin's handling of Java arrays*. Kotlin documentation. JetBrains. Jan. 21, 2025. URL: https://kotlinlang.org/docs/java-interop.html#java-arrays (visited on 04/03/2025).

[20] Kotlin programming language. *Let function*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/null-safety.html#let-function (visited on 04/05/2025).

[21] Kotlin programming language. *Mapped types*. Kotlin documentation. JetBrains. Jan. 21, 2025. URL: https://kotlinlang.org/docs/java-interop.html#mapped-types (visited on 04/03/2025).

[22] Kotlin programming language. *Not-null assertion operator*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/null-safety.html#not-null-assertion-operator (visited on 04/05/2025).

[23] Kotlin programming language. *Notation for platform types*. JetBrains documentation. JetBrains. Jan. 21, 2025. URL: https://kotlinlang.org/docs/java-interop.html#notation-for-platform-types (visited on 04/03/2025).

[24] Kotlin programming language. *Null safety*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/null-safety.html (visited on 04/05/2025).

[25] Kotlin programming language. *Null safety*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/java-to-kotlin-interop.html#null-safety (visited on 04/05/2025).

[26] Kotlin programming language. *Null-safety and platform types*. JetBrains documentation. JetBrains. Jan. 21, 2025. URL: https://kotlinlang.org/docs/java-interop.html#null-safety-and-platform-types (visited on 04/03/2025).

[27] Kotlin programming language. *Nullability annotations*. JetBrains documentation. JetBrains. Jan. 21, 2025. URL: https://kotlinlang.org/docs/java-interop.html#nullability-annotations (visited on 04/03/2025).

[28] Kotlin programming language. *Nullable receiver*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/null-safety.html#nullable-receiver (visited on 04/05/2025).

[29] Kotlin programming language. *Nullable types and non-nullable types*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/null-safety.html#nullable-types-and-non-nullable-types (visited on 04/05/2025).

[30] Kotlin programming language. *Package-level functions*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/java-to-kotlin-interop.html#package-level-functions (visited on 04/05/2025).

[31] Kotlin programming language. *Safe call operator*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/null-safety.html#safe-call-operator (visited on 04/05/2025).

[32] Kotlin programming language. *Safe casts*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/null-safety.html#safe-casts (visited on 04/05/2025).

[33] Kotlin programming language. *Set up a Kotlin/JS project*. Kotlin documentation. JetBrains. Apr. 4, 2025. URL: https://kotlinlang.org/docs/js-project-setup.html (visited on 04/05/2025).

[34] Kotlin programming language. *Static fields*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: https://kotlinlang.org/docs/java-to-kotlin-interop.html#static-fields (visited on 04/05/2025).

[35] 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

- 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

8.8 Presentation

- Line numbers for slides
- readability -> light mode