# Introduction to Kotlin

Christian Konersmann, Finn Paul Lippok, Paul Lukas
RWTH Aachen University, Aachen, Germany
{christian.konersmann,finn.lippok,paul.lukas}@rwth-aachen.de

Proseminar: Advanced Programming Concepts
Organiser: Prof. Dr. Jürgen Giesl
Supervisor: Jan-Christoph Kassing

April 11, 2025

**Abstract**

This paper introduces Kotlin, a statically typed, object-oriented programming language designed to be fully interoperable with Java and the Java Virtual Machine (JVM). It focuses on Kotlin's concise syntax, advanced features such as null safety, and extensive platform support enabled by its multiplatform development capability. These features make Kotlin a modern and powerful programming language.

## 1 Introduction

Kotlin is a modern programming language developed by JetBrains, a renowned software development company. Designed as a safer and more concise alternative to Java, Kotlin offers full interoperability with Java, allowing developers to leverage existing Java libraries and frameworks while benefiting from Kotlin's modern features. Its clean and expressive syntax has made it increasingly popular, particularly in Android development. In fact, Google announced Kotlin as its preferred language for Android app development in 2019 [3]. Beyond Android, Kotlin supports multiplatform development, enabling developers to build applications for the server, desktop, web, and iOS from a shared codebase [22]. This paper presents an overview of Kotlin's concise syntax and highlights key language features that illustrate its advantages over Java. It assumes a basic understanding of Java and begins by examining core syntactic differences between Kotlin and Java, followed by an introduction to concepts such as null safety and seamless Java interoperability. The paper concludes with a discussion of Kotlin's multiplatform capabilities, with particular emphasis on its application in Android development.

## 2 Basic Syntax

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

## 2.1 Program Entry Point and Method Declaration

The *main* method is the entry point of any Java or Kotlin program [8]. Java enforces object-oriented programming, thus requiring the *main* method to be declared within a class. For the *main* method to be directly executable, it must be *public* and *static*,[1] as shown below:

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 instantiate a class [20], functioning similarly to static methods in Java,[2] but without an explicit class affiliation. Kotlin further reduces boilerplate by having *public* as the default visibility [43] and allowing the *main* method to omit arguments passed as an array [8]. Also, semicolons are optional, and Kotlin introduces the *fun* keyword for defining functions [9], resulting in a concise and readable syntax, as shown here:

Kotlin main method

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

## 2.2 Variable Declaration

Variables in Kotlin are declared using the keyword *val* for immutable variables or *var* for mutable variables [10], similar to Java's *final* and non-final variables. The type 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 a variable's type based on its initializer or usage [2]. However, a detailed discussion of its internal workings is out of scope for this paper.

[1] If the *main* method were non-static, it would require an instance of the class before it could be run, leading to a circular dependency.

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

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

### 2.4 Return Type Declaration

Similar to variable declaration, a method's return type is declared after the method name and parameters, separated by a colon [9]. The equivalent of *void* in Java is *Unit* in Kotlin [1, 24], which can be omitted if no value is returned.

Java method declaration

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

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.[3] All types are objects and inherit from the *Any* class [11]. 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 treated as objects [26], enabling higher-order functions and functional programming paradigms. As a result, functions can be passed as arguments, returned from other functions, and assigned to variables.

## 3 Classes

In both Java and Kotlin, classes are declared using the *class* keyword [16] and can contain attributes, methods, and constructors. In the example below, we declare a class to represent a salesperson:

Java Class Declaration

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

```

[3] Certain types may be optimized to use Java primitives at runtime for performance reasons.

```
  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 [17]. As a reminder, the default visibility in Kotlin is *public*, and classes are also *final* by default, meaning they cannot be inherited from unless explicitly declared *open*. Kotlin also permits the declaration of constructor parameters (as properties) using *val* or *var*, including visibility modifiers [17], resembling the concise syntax found in 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 *salesVolume*. It is still possible to declare attributes outside the constructor. Additionally, Kotlin allows default parameter values in constructors and functions, a feature that would otherwise require method overloading in Java.

## 3.1 Properties

Properties in Kotlin facilitate concise declaration of getters and setters, including their visibility, directly beside the corresponding attribute [37, 38]. Unlike Java, accessing a property in Kotlin via dot notation will internally call the corresponding getter or setter method, ensuring a consistent 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 getters and setters can be defined inline [38]. The `field` keyword refers to the underlying attribute:

Custom accessors

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

Kotlin also supports computed properties, which do not store a value but compute it on access via a custom getter [38]. This approach mirrors writing a getter method in Java without a backing field:

Computed Property

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

## 4 String Interpolation

In Java, variables are typically formatted into strings using the *String.format()* method or by concatenation with the + operator. Kotlin introduces a more readable mechanism called string interpolation [42], allowing variables and expressions to be directly embedded within a string by prefixing them with a `$` sign, and enclosing expressions in curly braces:

String Interpolation

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

## 5 Null Safety

Whenever a method or attribute is accessed on a null reference in Java, a `NullPointerException` (NPE) is thrown. Kotlin eliminates this risk with a null safety system that distinguishes between nullable and non-nullable types [30, 35]. By default, all types in Kotlin are non-nullable, meaning variables cannot hold a null value unless explicitly declared with a question mark: Kotlin enforces this distinction at compile time, requiring developers to handle nullable types explicitly. At runtime, both types are treated the same.

```
var a: String = "a is non-nullable"
var b: String? = "b is nullable"
```

### 5.1 Null Safety Operators

When working with nullable types, you cannot directly access properties or methods, because the reference could be `null`, potentially causing an NPE. In Java, this is typically handled with an if check, as shown in the following example:

```
  private final SalesPerson supervisor;

  public void printSupervisor() {
    if (supervisor == null) System.out.println("null");
    else System.out.println(supervisor.name);
  }
```

This approach is also available in Kotlin, where checking for a null value in an if statement automatically casts the type to a non-nullable type within the scope of the statement [15].[4] However, Kotlin provides convenient shortcuts for handling nullable types by using Null safety operators.

[4]This feature is known as smart casting, where the compiler automatically casts the variable to a more specific type when it can guarantee the safety of the cast.

### 5.1.1 Safe Call Operator

The safe call operator `?.` allows you to access properties or methods of an object only if it is non-null, thereby reducing the need for explicit null checks [39]. If the object is `null`, the expression safely evaluates to `null`, avoiding an NPE. Otherwise, it evaluates as expected, granting access to the object's attributes or methods. Essentially, this operator enhances the standard dot notation by incorporating null safety.

```
val supervisor: SalesPerson? = null
fun printSupervisor() {
  println(supervisor?.name)
}
```

Multiple safe calls can be chained [39], and if any segment is `null`, the entire chain evaluates to `null`. Furthermore, Safe calls can also be used on the left side of an assignment. If the safe call operator evaluates to null, the assignment will be skipped.

```
// Chained safe call operators
var volume: Double? = supervisor?.supervisor?.salesVolume
// Assignment with a chained operator
supervisor?.supervisor?.salesVolume = 0.0
```

### 5.1.2 Elvis 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 evaluates to `null`, it returns a default value specified to the right side of `?:` [18]:

```
fun printSupervisor() {
  println(supervisor?.name ?: "No supervisor")
}
```

When compiling to Java, both the safe call operator and the Elvis operator are treated by the compiler as if statements. These operators simply make the code significantly shorter and easier to read, therefore increasing the maintainability as well.

### 5.1.3 Not-Null Assertion Operator

The not-null assertion operator (`!!`) converts a nullable type to a non-nullable type, instructing the compiler to treat the value as non-null [28]. However, if the value is actually `null`, a `NullPointerException` 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

```
var possiblyNull: String? = null
var b: String = possiblyNull!!
```

## 5.2 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 [34]. 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
```

# 6 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.

## 6.1 Call Java in Kotlin

Everything written in Java is accessible in Kotlin [13], 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.

### 6.1.1 Create and acces objects

To illustrate how Java code can be accessed in a Kotlin project, the following example features a Salesman class that stores basic information using getters and setters.

Example Java class

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

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

```
7       this.salary = salary;
8     }
9
10    public String getName() { return name; }
11    public int getSalary() { return salary; }
12    public void setSalary(int salary) { this.salary = salary; }
13  }
```

If we want to access this class, we can use the familiar Kotlin syntax [13] to instantiate the object and access its properties. There is no syntactical difference between accessing a Java class and a Kotlin class. Since there are no explicit getter and setter methods in Kotlin, Java methods following Java's conventions for getters and setters are converted [7] into so-called synthetic properties [23]. These can be accessed using Kotlin's property syntax. If the getters and setters do not follow Java conventions, they can still be accessed as regular methods.

Access a Salesman instance 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, therefore only a getter method will be created. 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 [23].

### 6.1.2 Mapped types

By default, when instances of a Java class are used in Kotlin, they are loaded as Java objects. However, some Java types have a corresponding Kotlin counterpart so the Java type is automatically replaced through the equivalent Kotlin type [27]. For example, `java.lang.Integer` is converted to `kotlin.Int?` because Java wrapper objects can be null. This applies to all Java wrapper classes and some important types, such as `java.lang.Object`, which is mapped to `kotlin.Any!`[5]. However, all Java primitive types are mapped to their non-nullable Kotlin counterparts, as primitive types cannot be null in Java. For instance, Java's `int` is converted to `kotlin.Int`. Additionally, collections like Lists, Maps and Arrays are also converted. For a complete list of all mapped types, consult the official documentation [27]. Java's return type void is replaced by Kotlin's Unit type.

### 6.1.3 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 with most Java types, the compiler assigns it a platform type, which is non-denotable. [32]. 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

[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 [29].

increases the risk of NullPointerExceptions. To demonstrate how this can be used in practice, the previously introduced Java Salesman class has been extended with the following method:

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

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.

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

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

Some Java compilers use annotations [33] to specify whether a value is nullable or non-nullable, such as JetBrains' @Nullable or @NotNull annotation [5]. If these annotations are present in the Java code, the compiler assigns the corresponding nullable or non-nullable 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
```

### 6.1.4 Java arrays in Kotlin

function 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 [25]. 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 [25]. This results in significant performance improvements compared to iterating over an `Array<Int>`, which would involve additional function calls and object overhead.

```kotlin
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 [25]. 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.

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

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

### 6.1.5 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 [19]. The following example demonstrates this by accessing a Java method named *in* from Kotlin.

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

## 6.2 Call Kotlin in Java

Just as Kotlin can create instances of Java classes, Java can also create and use instances of Kotlin classes [14].

### 6.2.1 Kotlin properties in Java

Kotlin properties cannot be accessed directly from Java and must be used via standard Java syntax. Therefore, the properties are compiled into a private field, along with corresponding getter and setter methods [6]. However if the Kotlin property is final, no setter method will be created. For example, consider a simple property in the SalesPerson class:

```kotlin
var name: String
```

This will compile to the following components in Java:

```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 preserves this visibility when generating the corresponding Java methods.

### 6.2.2 Null safety

If a public Kotlin function with a non-nullable parameter is called from Java, a nullable value can be passed to this function from Java. To retain null safety, Kotlin generates checks for those functions and throws a NullPointerException if the value is indeed null [31].

### 6.2.3 Package-level function

Package-level functions are functions, 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 [36] named by the Kotlin file the function oriented from, because in Java methods outside an class are prohibited. For example, consider a Kotlin file named `SalesPerson.kt` in the package `org.company`, containing the following 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");
```

The name of the generated class can be set using the `@file:JvmName("Example")` annotation in 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 [36].

### 6.2.4 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 to maintain code integrity. However, we can add the @JvmField annotation before our property to make it accessible in Java through dot notation [21]. 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) {}
```

[7] This does not apply to private properties.

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

Functions in companion or named objects can be marked as static to be accessed in Java using the same annotation [41], though this topic is beyond the scope of this paper.

### 6.3 Interoperability with JavaScript

Besides interoperability with Java, Kotlin also supports interoperability with JavaScript [40], 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 [4], 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 [40] provides a step-by-step setup guide. With this setup, it is possible to use JavaScript libraries or the DOM API [12] for web development using the familiar Kotlin syntax, thus making it a valid alternative to plain JavaScript. Furthermore, Gradle provides features that improve the workflow and simplify the development process.

## 7 Multiplatform development

Multiplatform development is a key benefit in Kotlin. By reusing the same code across multibile platforms we can save alot of time writing the code and can concentrate more on the differences between the platforms.

### 7.1 Hierarchical project structure

The Hierarchical project structure is best understood if we think of the source codes as objects with parent- and child-objects. These Objects have properties like the source code and targets. Targets are platforms to which Kotlin should be able to complie the code to and can also be tought of as tags. For example if we want to write an app for apple and androids. The common code used in bothe instances would be in the parent source which is called the commonMain and to which we then add the apple and android Tag to. So we could have the parent object commonMain which has the child objects andoridMain and appleMain in which we only have the specific Code for apple and android products and only have the corresponding tag. Further down the hierarchie we may differentiate between MacOS and IOS and even further down we could differentiate between different models. But as we declare more specific end targets we also need to add these specific tags to every parent object that is connected to the target In this case the appleMain and androidMain sources would be called intermediate source sets since they sit between the commonMain and the targets.

If we now wanted to compile code for the newest Iphone the tags become important, Kotlin would check every source set(object) for the right tag and compile them together to one source code. If a tag is placed wrong sothat the compiler cant reach the target from the commonMain source the compiler would ofcourse break.

### 7.2 Expected and actual declarations

Expected and actual declarations are somewhat similar to abstract functions. In a parent you can declare that this piece of code is expected further down the Hierarchical Structure and will be actualy declared there. Just like an abstract function which is declared in a parent object but gets overwritten by the cild object. So we could declare an expected Kotlin construct (function, class, interface ...) withe the "expect" keyword and use the refference in the common code. In the platform specific source we would need to declare the construct again and mark it as the actual construct with the "actual" keyword.

## 8 Android

This sections concentrates on the benefits Kotlin has in the Android enviorment not only the language itself but also Kotlin based Tools for Android

### 8.1 Jetpack Compose

Jetpack Compose is a Kotlin based Ui Tool Kit which eases the process of creating a Ui for Android Apps. By taking a different aproache to UI coding than XML the needed Code for UI can decrase by up to 50%. Eventough the size of the APK (Android Package) File and the build time will increase, these downsides are outweight by the increased productivity and maintainability of the Code.

In the following cases we boththimes just print Hello World to the Ui

XML Hello World

```
<TextView
  android:id="@+id/textView"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Hello␣World" />
```

Jetpack Compose Hello World

```
@Composable
fun SimpleText() {
  Text(text = "Hello␣World")
}
```

As we can see the readeability and the needed Code is improved alot in Jetpack Compose

Jetpack Compose achieves this by letting the user descripe what the UI should look like in a given state rather then how the UI should be build. This becomes especially usefull when the state changes. If the state changes Jetpack Compose automatically updates and rerenders the important parts. If we wouldnt use Jetpack Compose we would have to maually find the views, update them and manage the state changes ourselfes.

Jetpack Compose also combines the UI and the Logic. In XMl we can only define the look of for example a button but not the Logic that follows clicking that button. For the Logic we need a sepreate Tool like Activity, there we can define the Logic of the button. But since XML and Activity are two unrealted tools you have to connect the code mannualy by using the findViewById command.

```
val button = findViewById<Button>(R.id.button)
```

so a simple Button counting up would look in XML and Activity something like:

XML UI Button

```
<Button
  android:id="@+id/button"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Clicked␣0␣times" />
```

Activity Logic Button

```
var count = 0
val button = findViewById<Button>(R.id.button)
button.setOnClickListener {
  count++
  button.text = "Clicked␣$count␣times"
}
```

Since Jetpack Compose combines UI and Logic the Kotlin code only looks like the following

Jetpack Compose Button

```
@Composable
fun ClickButton() {
  var count by remember { mutableStateOf(0) }
  Button(onClick = { count++ }) {
    Text("Clicked␣$count␣times")
  }
}
```

As you might have noticed in the JC Code samples we always write the @Composable Annotation this is very Important so the Compiler knows which functions are meant to be UI elements and should be updated depending on the state. If we tried to create a UI element without the @Composable Annotation it wouldnt be treated as a UI element and if we used a UI function like Text() in our element we would get a Compiler Error.

## 8.2 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. Multithreading is very common and basicaly means, that we split up the processeing among multibile threads that we can run parallel. You could imagine the threads like computers running at the same time. If one of them crashes the others are not affected and since they are running at the same time it is in many cases alot faster then if we only had one computer running everything. As you also couldnt fit unlimited computers in one room, the useage of threads is mainly limited by the amount of memmory the programm is allowed to use.

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, the equivalent to 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 { //creates a new coroutine
  val sales = getSalesVolume()
  withContext(Dispatchers.Main) { //switches to main thread
    textView.text = sales
  }
}
```

Kotlin achieves this by using one thread to run multibile coroutines. So if you had a piece of code that pauses inbetween, Kotlin would run a differrent coroutine while the first one is paused on the same thread thus not having to open up more threads and saving alot of memmory. By managing the couroutines in Kotlin they can switch extremly fast between coroutines in one thread instead of switching the context in threads which are mangaed by the OS. So coroutines are not really a new version of threads but a system that cleverly manages the resources given to minimize the unnesesary use of Threads. So this would mean if we had really long coroutines that have no pauses in between, Kotlin couldnt run multibile coroutines on one thread and would have to open up multible threads thus losing the advantage of using coroutines. Kotlin also keeps the threads if they are not in use and sorts them in to pools for different usage(IO,Default,Main), so that it can later check these pools for already setup but unused therads . So if you create the first coroutine Kotlin still has to setup a thread but after that Kotlin will reuse that thread as often as it can before it is forced to create a new one. Coroutines also setup very fast, if we already had coroutines setup prior, since we dont have to setup a new thread but are just reusing one that is already there.

## 8.3 Android KTX

Android KTX is a collection of Kotlin-friendly libraries that sit on top of the existing Android APIs. It doesn't replace the Android SDK, but simplifies and enhances it to work better with Kotlin by for example using Coroutines instead of Threads. In the following example we want to save the revenue of a salesman.

Java save salesman revenue

```
SharedPreferences prefs =
    PreferenceManager.getDefaultSharedPreferences(context);
SharedPreferences.Editor editor = prefs.edit();
editor.putFloat("salesman_revenue", 1234.56f);
editor.apply();
```

Kotlin save salesman revenue

```
context.defaultSharedPreferences.edit {
  putFloat("salesman_revenue", 1234.56f)
}
```

# 9 Conclusion

Kotlin is a modern programming language that builds upon Java's foundation, offering a concise syntax, improved class structures, and innovative features such as null safety. Its seamless interoperability with Java and JavaScript makes it an excellent choice for projects requiring integration with existing codebases. Kotlin's support for multiplatform development, particularly in Android, positions it as a powerful tool for developers seeking to create cross-platform applications.

While this paper provides an introduction to Kotlin, it only scratches the surface of its capabilities. Advanced features such as smart casts, delegation, and destructuring declarations further enhance Kotlin's appeal. Kotlin also embraces functional programming paradigms, inspired by languages like Haskell. It supports higher-order functions, lambda expressions, and immutability, enabling developers to write expressive and concise code. These features allow developers to adopt a functional style while still benefiting from Kotlin's object-oriented capabilities.

These features, combined with its modern design and developer-friendly syntax, make Kotlin a compelling choice for both new and experienced developers.

# 10 TODO

## 10.1 General

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

## 10.2 Introduction

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

## 10.3 Basic Syntax

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

## 10.4 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?)

## 10.5 New Features

- 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)

## 10.6 Multiplatform development

## 10.7 Android

- Discuss Kotlin's advantages for Android development

## 10.8 Presentation

- Line numbers for slides
- readability -> light mode

# 11 TODO

## 11.1 General

- Add citations
- Fix formatting (especially indentation in the code snippets)
- check reference title, some are not the same as in the website

## 11.2 Basic Syntax

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

## 11.3 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?)

## 11.4 New Features

- 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)

## 11.5 Multiplatform development

## 11.6 Android

- Discuss Kotlin's advantages for Android development

## 11.7 Presentation

- Line numbers for slides
- readability -> light mode