

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 12, 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 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 [4]. Beyond Android, Kotlin supports multiplatform development, enabling developers to build applications for the server, desktop, web, and iOS from a shared codebase [27]. 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 [7]. 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

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

Kotlin, on the other hand, does not require methods to be declared inside a class, allowing for a more functional programming style with top-level functions. These top-level functions can be called directly without the need to instantiate a class [26], 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 [40] and allowing the *main* method to omit arguments passed as an array [7]. Also, semicolons are optional, and Kotlin introduces the *fun* keyword for defining functions [8], resulting in a concise and readable syntax, as shown here:

Kotlin main method

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

2.2 Variable Declaration

Variables in Kotlin are declared using the keyword *val* for immutable variables or *var* for mutable variables [9], similar to Java's *final* and non-final variables. The type is declared after the variable name, separated by a colon:

Java data types

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

Kotlin data types

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

2.3 Type Inference

Kotlin supports type inference, allowing the compiler to infer a variable's type based on its initializer or usage [3]. Unlike many functional programming languages such as Haskell,

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

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

which rely on unification for type inference, Kotlin uses a different approach called constraint solving [2]. This approach allows for type inference with subtyping. However, a detailed discussion of its internal workings is out of scope for this paper.

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

2.4 Return Type Declaration

Similar to variable declaration (Sect. 2.2), a method’s return type is declared after the method name and parameters, separated by a colon [8]. The equivalent of *void* in Java is *Unit* in Kotlin [1, 28], which can be omitted if no value is returned.

Java method declaration

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

Kotlin method declaration

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

2.5 Everything is an Object

In Kotlin, there are no primitive types.³ All types are objects and inherit from the *Any* class [10]. This approach creates a more consistent object-oriented programming model and eliminates the need for wrapper classes.

Java Integer Wrapper

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

Kotlin direct usage of Int

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

Furthermore, functions in Kotlin are also treated as objects [29], 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 [23] and can contain attributes, methods, and constructors. In the example below, we declare a class to represent a salesperson:

Java Class Declaration

```
1 public class SalesPerson {
2     private final String name;
```

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

```

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

```

Kotlin improves upon Java by allowing the constructor to be declared directly within the class definition [24]. 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 [24], resembling the concise syntax found in Java records:

Kotlin Class Declaration

```

1  class SalesPerson(val name: String, private val commissionRate:
2      Int, transferAmount: Double = 0.0) {
3      var salesVolume: Double = transferAmount
4  }

```

In this example, *name* and *commissionRate* become properties of the class, while *transferAmount* is a constructor parameter used to initialize *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

```

1  var salesVolume: Double = transferAmount
2      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

```

1  var salesVolume: Double = transferAmount
2      private set(value) {
3          if (value < 0)
4              throw IllegalArgumentException("SV must be positive")
5          field = value
6      }

```

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

```
1 val commission: Double
2     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 [39], 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

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

5 Extension Functions

Extension functions enable the declaration of new methods for a class without modifying the class's source code [25]. Data encapsulation remains intact, as extension functions can only access the public members of the class. This feature is particularly useful when adding utility methods to existing classes you cannot modify, such as third-party libraries:

Extension Function

```
1 // Extension function outside of the class
2 fun SalesPerson.print() {
3     println("$name: $salesVolume sold")
4 }
5
6 fun main() {
7     val carl = SalesPerson("Carl", 1200)
8     carl.print() // Calling the extension function
9 }
```

6 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, 31]. 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.

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

6.1 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:

```
1 private final SalesPerson supervisor;  
2  
3 public void printSupervisor() {  
4     if (supervisor == null) System.out.println("null");  
5     else System.out.println(supervisor.name);  
6 }
```

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 [32].⁴ However, Kotlin provides convenient shortcuts for handling nullable types by using Null safety operators.

6.1.1 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 [33]. 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.

```
1 val supervisor: SalesPerson? = null  
2 fun printSupervisor() {  
3     println(supervisor?.name)  
4 }
```

Multiple safe calls can be chained [33], 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.

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

6.1.2 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 `?:` [34]:

```
1 fun printSupervisor() {  
2     println(supervisor?.name ?: "No supervisor")  
3 }
```

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.

⁴This 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.

6.1.3 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 [35]. 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

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

6.2 Nullable Receiver

Nullable receivers refer to extension functions (Sect. 5) that can be called on nullable objects [36]. A normal extension function cannot be called on a nullable type, as the compiler cannot guarantee that the object is not null. However, it is possible to define an extension function that can be called on nullable types by implementing a so-called *nullable receiver*, which is indicated by a question mark after the class name in the method declaration. By handling the null case within the method itself, the method remains accessible even if the object is null. The following example demonstrates how to define and properly use an extension function with a nullable receiver type.

Usage of an extension function

```
1 // define the extension function
2 fun SalesPeron?.print() {
3     if (this == null) return println("This person does not exist.")
4     return println("$name: $salesVolume sold")
5 }
6 // use the extension function
7 var sales: SalesPeron? = null
8 sales.print() // This person does not exist.
9 sales = SalesPeron("Carl", 1200)
10 sales.print() // Carl: 0.0 sold
```

7 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.

7.1 Call Java in Kotlin

Everything written in Java is accessible in Kotlin [11], making interoperability especially useful when working with one of the countless Java libraries, 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.

7.1.1 Create and access objects

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

Example Java class

```
1 public class Salesman {
2     private final String name;
3     private int salary;
4
5     public Salesman(String name, String title, int salary) {
6         this.name = name;
7         this.salary = salary;
8     }
9
10    public String getName() { return name; }
11    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 [11] to instantiate the object and access its properties. There is no syntactical difference between accessing a Java class and a Kotlin class, since Java methods following Java's conventions for getters and setters are converted [12] into so-called synthetic properties [13]. 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 [13].

7.1.2 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, these objects are automatically mapped to their equivalent Kotlin type [14]. For example, `java.lang.Integer` is converted to `kotlin.Int`⁵ 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 exhaustive list of all mapped types, consult the official documentation [14]. Java's return type `void` is replaced by Kotlin's `Unit` type.

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

7.1.3 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. [15]. This means we cannot explicitly declare this type as we do with nullable types using a question mark⁶. With platform types, Kotlin relaxes its strict null safety rules, making their handling similar to Java. However, this increases the risk of `NullPointerException`s. To demonstrate how this can be used in practice, the previously introduced Java Salesman class has been extended 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 the type is set to non-nullable but the object is actually null, attempting to access it will result in a `NullPointerException`, as shown above. Therefore, using nullable types is generally safer.

Some Java compilers use annotations [17] 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. For example, if a method in the Salesman class returns a string annotated with `@NotNull`, the variable would be assigned a non-nullable type instead of a platform type:

```
1 public static @NotNull String getString() { return "Not_null"; }
```

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

7.1.4 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 [18]. The compiler optimizes the code and uses primitive arrays whenever possible. For example, Java's `int[]`

⁶When the compiler needs to report a type-related error, it uses an exclamation mark to indicate the platform type [16].

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:

```
1 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 [18]. This results in significant performance improvements compared to iterating over an `Array<Int>`, which would involve additional function calls and object overhead.

```
1 var array: IntArray = intArrayOf(1, 2, 3)
2 takeArray(array) // passes int[] to Java function
3 for (i in array.indices) // no iterator created
4     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 does not allow this, thus there arrays are invariant [18]. However there is an exception when you need to parse an array to Java, it is allowed for platform types, because Java treats arrays as covariant. If we had a method, that requires an Object array, we could parse a string array of platform type to it.

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

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

7.2 Call Kotlin in Java

Just as Kotlin can create instances of Java classes, Java can also create and use instances of Kotlin classes [19].

7.2.1 Kotlin properties in Java

Kotlin properties cannot be directly accessed from Java, and must therefore be compiled into private fields, along with their corresponding getter and setter methods [6]. However, if the Kotlin property is final, no setter method will be created. For example, consider this simple property in the `SalesPerson` class:

```
1 var name: String
```

This will compile to the following components in Java:

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

The Kotlin compiler will preserve the restricted visibility of setters and getters when generating the corresponding Java methods.

7.2.2 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 [20].

7.2.3 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 Kotlin⁷. This example demonstrates how to use the annotation.

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

1 public void example() {
2     SalesPerson person = new SalesPerson("Carl");
3     System.out.println(person.name); // prints 'Carl'
4 }
```

Functions in companion or named objects can be marked as static to be accessed in Java using the same annotation [22], though this topic is beyond the scope of this paper.

8 Multipatform development

Kotlin can be compiled not only for the JVM but also into native binaries, removing the need for a virtual machine as required by Java. This makes Kotlin suitable for use in embedded systems, where the overhead of running a virtual machine is impractical. In addition to Java, Kotlin is also interoperable with C as well as Swift/Objective-C and can be compiled to pure JavaScript or WebAssembly. Because Kotlin can be compiled to multiple targets, it is well suited for multipatform development. To further simplify the process of writing code for multiple applications across various platforms, Kotlin offers a hierarchical project structure. This structure allows the definition of a shared codebase that can be reused across platform-specific subprojects. Each subproject can integrate this shared code with its platform-specific libraries, improving code maintainability and minimizing redundancy.

Leveraging Kotlin's Multipatform programming support, you can develop applications for servers, desktop, web, and mobile, making it a key advantage of the Kotlin programming language.

9 Android

This section focuses on the benefits Kotlin offers in the Android environment, not only through the language itself but also through Kotlin-based tools for Android.

9.1 Jetpack Compose

Jetpack Compose is a Kotlin-based UI toolkit that simplifies Android UI development. Unlike XML, the traditional method for designing Android UIs, it can reduce required

⁷This does not apply to private properties.

code by up to 50%. While it may slightly increase APK size and build time, the gains in productivity and maintainability far outweigh these downsides. Compose improves readability and handles UI updates automatically based on state changes, so there is no need to manually update views or manage the states. Jetpack Compose also integrates logic seamlessly. Unlike XML, where logic must be handled separately, Compose keeps everything unified. All UI functions must be annotated with `@Composable`, which informs the compiler to treat them as UI elements that react to state. Without it, using elements like `Text()` would cause a compiler error.

9.2 Android KTX

Android KTX is a collection of Kotlin-friendly libraries that build on top of the existing Android APIs, enhancing the development process by providing Kotlin-friendly APIs, such as Coroutines.

9.3 Coroutine

Imagine we want an app that shows a salesperson's live sales. Checking this every second would freeze the app unless we use multithreading, this means splitting tasks across parallel threads, threads are subprocesses which can run independent from the main process, thus being able to freeze without stopping the main thread. This speeds things up and keeps the app responsive, though memory limits how many threads we can run.

In Kotlin, we use coroutines instead of traditional threads. Coroutines are lightweight, easy to read, and far more memory-efficient than Java threads, so we can track thousands of salespeople without crashing the app. Kotlin runs multiple coroutines on a single thread by pausing one and switching to another, all within the same thread. This avoids the heavy context switching done by the OS with threads. If a coroutine runs too long without pausing, Kotlin must spin up new threads, which loses the performance benefits. To optimize even further, Kotlin reuses threads from internal pools (e.g., IO, Default, Main). The first coroutine starts a thread, but after that, existing threads are reused, making coroutine setup extremely fast.

Coroutines work on all platforms but are especially useful in apps, which often run on resource-limited devices and handle frequent data transfers that pause execution. Their lightweight nature makes them ideal for this. On Android, features like `lifecycleScope` (from Jetpack ([Sect. 9.1](#))) help manage coroutines by automatically canceling them when an activity is destroyed, preventing memory leaks.

10 Conclusion

Kotlin is a modern programming language that offers a concise syntax, improved class structures, and innovative features such as null safety. Its seamless interoperability with Java makes it a great choice for projects integrating with existing codebases and libraries. Kotlin's support for multiplatform development, particularly in Android, establishes it as a powerful option for cross-platform development.

While this paper provides a solid foundation, it only scratches the surface of Kotlin's capabilities. Advanced features such as smart casts, delegation, and destructuring declarations further enhance Kotlin's appeal. By also embracing functional programming paradigms inspired by languages like Haskell, Kotlin enables developers to write cleaner and more maintainable code while still benefiting from its object-oriented capabilities.

These features, combined with its modern design and developer-friendly syntax, make Kotlin a powerful alternative to Java and a compelling choice for developers.

References

- [1] Marat Akhin and Mikhail Belyaev. *Built-in types and their semantics*. *Kotlin.Unit*. Kotlin specification. JetBrains. URL: <https://kotlinlang.org/spec/built-in-types-and-their-semantics.html#kotlin.unit> (visited on 04/08/2025).
- [2] Marat Akhin and Mikhail Belyaev. *Kotlin Type Constraints*. Kotlin specification. JetBrains. URL: <https://kotlinlang.org/spec/kotlin-type-constraints.html#kotlin-type-constraints> (visited on 04/11/2025).
- [3] Marat Akhin and Mikhail Belyaev. *Type Inference*. Kotlin specification. JetBrains. URL: <https://kotlinlang.org/spec/type-inference.html#type-inference> (visited on 04/08/2025).
- [4] Chet Haase. *Google I/O 2019: Empowering developers to build the best experiences on Android + Play*. Android developers blog. Google. May 7, 2019. URL: <https://android-developers.googleblog.com/2019/05/google-io-2019-empowering-developers-to-build-experiences-on-Android-Play.html> (visited on 04/09/2025).
- [5] 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).
- [6] Kotlin pro2024-12-16gramming language. *Calling Kotlin from Java. Properties*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/java-to-kotlin-interop.html#properties> (visited on 04/05/2025).
- [7] 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).
- [8] Kotlin programming language. *Basic Syntax. Functions*. Kotlin documentation. JetBrains. Nov. 6, 2024. URL: <https://kotlinlang.org/docs/basic-syntax.html#functions> (visited on 04/08/2025).
- [9] Kotlin programming language. *Basic Syntax. Variables*. Kotlin documentation. JetBrains. Nov. 6, 2024. URL: <https://kotlinlang.org/docs/basic-syntax.html#variables> (visited on 04/08/2025).
- [10] Kotlin programming language. *Basic Types*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/basic-types.html> (visited on 03/29/2025).
- [11] 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).
- [12] Kotlin programming language. *Calling Java from Kotlin. Getters and setters*. JetBrains documentation. JetBrains. Jan. 6, 2025. URL: <https://kotlinlang.org/docs/java-interop.html#getters-and-setters> (visited on 04/21/2025).
- [13] Kotlin programming language. *Calling Java from Kotlin. 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).
- [14] Kotlin programming language. *Calling Java from Kotlin. Mapped types*. Kotlin documentation. JetBrains. Jan. 21, 2025. URL: <https://kotlinlang.org/docs/java-interop.html#mapped-types> (visited on 04/03/2025).

- [15] Kotlin programming language. *Calling Java from Kotlin. 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).
- [16] Kotlin programming language. *Calling Java from Kotlin. 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).
- [17] Kotlin programming language. *Calling Java from Kotlin. Nullability annotations*. JetBrains documentation. JetBrains. Jan. 21, 2025. URL: <https://kotlinlang.org/docs/java-interop.html#nullability-annotations> (visited on 04/03/2025).
- [18] Kotlin programming language. *Calling Java from Kotlin. Java arrays*. Kotlin documentation. JetBrains. Jan. 21, 2025. URL: <https://kotlinlang.org/docs/java-interop.html#java-arrays> (visited on 04/03/2025).
- [19] Kotlin programming language. *Calling Kotlin from Java*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/java-to-kotlin-interop.html> (visited on 04/05/2025).
- [20] Kotlin programming language. *Calling Kotlin from Java. 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).
- [21] Kotlin programming language. *Calling Kotlin from Java. 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).
- [22] Kotlin programming language. *Calling Kotlin from Java. 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).
- [23] Kotlin programming language. *Classes*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/classes.html> (visited on 04/08/2025).
- [24] Kotlin programming language. *Classes. Constructors*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/classes.html#constructors> (visited on 04/08/2025).
- [25] Kotlin programming language. *Extensions*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/extensions.html> (visited on 04/12/2025).
- [26] 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).
- [27] Kotlin programming language. *Introduction to Kotlin Multiplatform*. Kotlin documentation. JetBrains. Dec. 16, 2024. URL: <https://kotlinlang.org/docs/multiplatform-intro.html> (visited on 04/09/2025).
- [28] Kotlin programming language. *Kotlin Standard Library. Kotlin.Unit*. Kotlin API documentation. JetBrains. URL: <https://kotlinlang.org/api/core/kotlin-stdlib/kotlin/-unit/> (visited on 04/08/2025).
- [29] Kotlin programming language. *Lambdas. Higher-order functions*. Kotlin documentation. JetBrains. Nov. 27, 2024. URL: <https://kotlinlang.org/docs/lambdas.html#higher-order-functions> (visited on 04/08/2025).

- [30] Kotlin programming language. *Null safety*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/null-safety.html> (visited on 04/05/2025).
- [31] Kotlin programming language. *Null safety. 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).
- [32] Kotlin programming language. *Null safety. 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).
- [33] Kotlin programming language. *Null safety. 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).
- [34] Kotlin programming language. *Null safety. Elvis operator*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/null-safety.html#elvis-operator> (visited on 04/05/2025).
- [35] Kotlin programming language. *Null safety. 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).
- [36] Kotlin programming language. *Null safety. Nullable receiver*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/null-safety.html#nullable-receiver> (visited on 04/05/2025).
- [37] Kotlin programming language. *Properties. Declaring properties*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/properties.html#declaring-properties> (visited on 04/08/2025).
- [38] Kotlin programming language. *Properties. Getters and setters*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/properties.html#getters-and-setters> (visited on 04/08/2025).
- [39] Kotlin programming language. *Strings in Java and Kotlin. String concatenation*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/java-to-kotlin-idioms-strings.html#concatenate-strings> (visited on 04/08/2025).
- [40] Kotlin programming language. *Visibility Modifiers*. Kotlin documentation. JetBrains. Sept. 25, 2024. URL: <https://kotlinlang.org/docs/visibility-modifiers.html> (visited on 03/24/2025).