Android Development

Kotlin

Mark Joselli

mark.joselli@pucpr.br

Kotlin

Kotlin

- Kotlin is a modern, statically typed programming language developed by JetBrains that is fully interoperable with Java and designed to be a more concise, safe, and expressive language.
- It is officially supported by Google for Android application development and offers several advantages compared to Java.

Google x Oracle

- The relationship between Google and Oracle played a significant role in Google's decision to adopt Kotlin as an official programming language for Android app development.
- This relationship was marked by prolonged legal disputes over the use of Java on Android.

Google x Oracle – Sun

- In 2010, Oracle acquired Sun Microsystems, the original company that developed the Java programming language. With this acquisition, Oracle became the owner of the rights to Java.
- Shortly after the acquisition, Oracle filed a lawsuit against Google, alleging that Android used parts of Java in an unauthorized manner, specifically the Java API.
- Oracle alleged that Google violated its copyright and patent rights by using 37 Java API packages to develop Android.

Google x Oracle - Battle

- The case went through several phases of trials and appeals.
- In some trials, Google won, with courts ruling that the use of Java APIs was covered by "fair use."
- In other instances, decisions were in favor of Oracle.
- The long legal battle has created uncertainty in the developer community about the future of using Java in Android development.
- There were concerns about possible legal repercussions and costs associated with using Java.

Google x Oracle - Uncertain

- Faced with legal disputes and uncertainty associated with the use of Java, Google began looking for alternatives.
- Kotlin, with its modern features and interoperability with Java, has emerged as a viable solution.
- Kotlin offered the opportunity to continue developing Android applications without being directly tied to the legal complexities associated with Java.

Google x Oracle – End

- In April 2021, the United States Supreme Court ruled in favor of Google, stating that Google's use of the Java APIs was considered "fair use."
- This decision brought an end to the long legal battle between Google and Oracle.
- The decision was seen as a significant victory for Google and the developer community, as it set an important precedent regarding the use of APIs in software.

Main Features of Kotlin

- Concise and Expressive Syntax:
  - Less Verbosity: Kotlin significantly reduces the amount of boilerplate code compared to Java.
  - Type Inference: The compiler can infer variable types automatically, reducing the need for explicit declarations.
- Interoperability with Java:
  - Full Interoperability: Kotlin can call and be called by Java code, facilitating the gradual migration of existing Java projects.
  - Libraries and Frameworks: You can use all Java libraries and frameworks, which allows you to reuse existing code.

Main Features of Kotlin

- Security and Reliability:
  - Null Safety: Kotlin's type system eliminates most null pointer errors at compile time, helping to prevent the most common cause of crashes on Android.
  - Immutability: Promotes the use of immutable classes and variables, which helps create more robust and easier to maintain code.
- Functional Programming:
  - Higher Order Functions: Support for functions that receive or return other functions.
  - Lambdas: Lambda expressions and anonymous functions are supported, facilitating functional programming.
- Coroutines:
  - Asynchronous Programming: Facilitates writing asynchronous code by allowing the use of coroutines for long-running operations without blocking the main thread.

Site

- https://kotlinlang.org/

Basic Syntax and Type Inference

- In Kotlin, you can define mutable variables with var and immutable variables with val.
- Type inference allows the compiler to deduce the type of the variable automatically.

```
val name = "Alice"    // constant
var age = 30          // variable
```

Kotlin types

- Kotlin offers a variety of types that are well integrated with Java's type system, as well as supporting primitive types, collections, and user-defined types. Here is a summary of types in Kotlin with examples:
- Primitive Types
- Kotlin automatically converts primitive types to their corresponding Java wrapper classes when necessary. The main primitive types are:
- Int
- Long
- Short
- Byte
- Float
- Double
- Char
- Boolean

```
val myInt: Int = 42
val myLong: Long = 3000000000L
val myFloat: Float = 3.14F
val myDouble: Double = 3.14
val myChar: Char = 'A'
val myBoolean: Boolean = true
```

Strings

- Strings are immutable and can be manipulated using various extension functions.

```
val myString: String = "Hello, World!"
println(myString.length)
println(myString.uppercase())
```

Arrays

- Arrays are supported as part of the Kotlin standard library and can be created in several ways.

```
val myArray: Array<Int> = arrayOf(1, 2, 3, 4, 5)
val anotherArray = intArrayOf(1, 2, 3, 4, 5)
println(myArray.size)
println(anotherArray[2])
```

Collections

- Kotlin has a rich collection library that includes lists, sets, and maps, both mutable and immutable.

```
// imutable list
val myList: List<String> = listOf("One", "Two", "Three")

// mutable list
val myMutableList: MutableList<String> = mutableListOf("One", "Two", "Three")
myMutableList.add("Four")

// set
val mySet: Set<Int> = setOf(1, 2, 3, 4, 5)

// map
val myMap: Map<String, Int> = mapOf("Alice" to 30, "Bob" to 25)
val myMutableMap: MutableMap<String, Int> = mutableMapOf("Alice" to 30, "Bob" to 25)
myMutableMap["Charlie"] = 35
```

```
// one line comment
/*
* comment in multiple lines
* ok.
*/
```

Comments

- Comments in Kotlin are similar to those in other programming languages such as Java and C#.
- There are two main types of comments: single-line comments and multi-line comments.
- Common tags in documentation comments include:
  - @param: Describes a function or method parameter.
  - @return: Describes the return value of a function or method.
  - @throws or @exception: Describes an exception that a function or method can throw.
  - @author: Informs the author of the code.
  - @see: Provides a reference to another element of the code.

Flow Control Structures
Conditionals (if/else)

- The if in Kotlin can be used both as an expression and a statement. This means it can return a value.

```
if (a > b) {
    println("a é maior que b")
} else {
    println("a não é maior que b")
}

val max = if (a > b) a else b
```

When

- The when is a more powerful replacement for the switch found in other languages. It can be used both as an expression and as a statement.

```
when (x) {
    1 -> println("x é 1")
    2 -> println("x é 2")
    else -> println("x não é nem 1 nem 2")
}

val result = when (x) {
    1 -> "Um"
    2 -> "Dois"
    else -> "Desconhecido"
}

val obj: Any = "Hello"

val result = when (obj) {
    is String -> "O objeto é uma String com tamanho
${obj.length}"
    is Int -> "O objeto é um Int com valor $obj"
    else -> "Tipo desconhecido"
}
```

loops - FOR

```
val items = listOf("apple", "banana", "kiwifruit")
for (item in items) {
    println(item)
}

for (index in 1..5) {
    println("Index: $index")
}


for ((index, value) in items.withIndex()) {
    println("O item na posição $index é $value")
}

for (i in 1..10 step 2) {
    print("$i ")
}


for (i in 10 downTo 1) {
    print("$i ")
}
```

While & Do-While

```
// while loop
while (y > 0) {
    println(y)
    y--
}

// do-while loop
var z = 5
do {
    println(z)
    z--
} while (z > 0)
```

Continue & Break

```
for (i in 1..10) {
    if (i == 3) break
    println(i)
}

for (i in 1..10) {
    if (i == 3) continue
    println(i)
}
```

Functions

- Functions in Kotlin are a fundamental part of the language and offer a variety of advanced features that help you write concise and expressive code.
- The definition of a function in Kotlin starts with the fun keyword, followed by the function name, a list of parameters in parentheses, and the return type.

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

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

fun greet(name: String = "World") {
    println("Hello, $name!")
}
```

Extensions

- Extension functions allow you to add new functionality to existing classes without modifying them.

```
fun String.greet() {
    println("Hello, $this!")
}

fun main() {
    "Kotlin".greet()
}
```

Lambda Functions and Higher Order Functions

- Lambdas are anonymous functions that can be treated as values. They are often used in higher order functions.
- Higher-order functions accept other functions as parameters or return functions.

```
fun calculate(x: Int, y: Int, operation: (Int, Int) -> Int): Int {
    return operation(x, y)
}

val sum = calculate(5, 3) { a, b -> a + b }

val printMessage = { message: String -> println(message) }
printMessage("Hello from Lambda")
```

Functions Inline

- Inline functions can be used to improve performance by avoiding creating function objects at runtime.

```
inline fun performOperation(x: Int, y: Int, operation: (Int, Int) -> Int): Int {
    return operation(x, y)
}

val result = performOperation(5, 3) { a, b -> a + b }
```

Local functions

- Functions can be nested inside other functions, which is useful for specific logic that doesn't need to be reused elsewhere.

```
fun outerFunction() {
    fun innerFunction() {
        println("Inner function")
    }

    innerFunction()
}

outerFunction()
```

Function with Vararg Parameter

```
fun printAll(vararg messages: String) {
    for (message in messages) println(message)
}

fun main() {
    printAll("Hello", "World", "Kotlin")
}
```

Class

- In Kotlin, a base class is defined with the class keyword.
- To create an instance of a class we just call the constructor.

```
class Person {
    var name: String = ""
    var age: Int = 0
}

val person = Person()
person.name = "Alice"
person.age = 30
```

Constructor

- Kotlin supports primary and secondary constructors.
- Primary Constructor
- The primary constructor is defined in the class declaration itself.
- Secondary Builders
- Secondary constructors are defined within the class body.

```
class Person(val name: String, var age: Int) {
    var address: String = ""

    constructor(name: String, age: Int, address: String) :
this(name, age) {
        this.address = address
    }
}
```

Methods

- Methods are functions defined within a class.
- public (default): The public modifier makes the member visible everywhere. This is the default modifier if no other is specified.
- Internal: The internal modifier makes the member visible within the same module (compilation).
- Protected: The protected modifier makes the member visible only within the class and subclasses.
- Private: The private modifier makes the member visible only within the class where it was declared.

```
class Person(private val name: String, protected var age:
Int) {

    public fun greet() {
        println("Hello, my name is $name and I am $age
years old.")
    }


    protected fun protectedMethod() {
        println("This is a protected method.")
    }


    private fun privateMethod() {
        println("This is a private method.")
    }


    internal fun internalMethod() {
        println("This is an internal method.")
    }
}
```

Heritage

- Kotlin supports inheritance between classes with the open keyword.

```kotlin
open class Animal(val name: String) {
    open fun sound() {
        println("Animal sound")
    }
}

class Dog(name: String) : Animal(name) {
    override fun sound() {
        println("Bark")
    }
}

fun main() {
    val dog = Dog("Buddy")
    dog.sound()  // Output: Bark
}
```

Abstract Classes

- Abstract classes cannot be instantiated and can contain abstract methods that must be implemented by subclasses.

```
abstract class Animal(val name: String) {
    abstract fun sound()
}

class Cat(name: String) : Animal(name) {
    override fun sound() {
        println("Meow")
    }
}
```

Interfaces

- Interfaces can contain declarations of abstract methods and properties, as well as default implementations.

```
interface Drivable {
    fun drive()
}

class Car : Drivable {
    override fun drive() {
        println("Driving a car")
    }
}
```

Data Classes

- Data classes are used for classes that are mainly used to store data.

```
data class User(val id: Int, val name: String)
```

Object Declarations

- Kotlin supports singleton objects using the object keyword.

```
object Database {
    fun connect() {
        println("Connected to database")
    }
}

fun main() {
    Database.connect()  // Output: Connected to database
}
```

Companion Objects

- Companion objects allow you to define members that belong to the class, not class instances.

```
class MyClass {
    companion object {
        fun create(): MyClass = MyClass()
    }
}

fun main() {
    val instance = MyClass.create()
}
```

Exercise

List of kotlin exercises.