KAMI LIMU

19th June 2024

Introduction to Kotlin

Introduction to the Kotlin programming language

KAMI LIMU

What we will cover

- What is Kotlin
- Why Kotlin for Android development
- Kotlin features
- Unit testing

What is Kotlin?

- A modern, statically typed OOP language that runs on the Java Virtual Machine (JVM)
- Developed by JetBrains
- Kotlin is designed to be fully interoperable with Java
- Is open source
- Can be compiled into JavaScript

Why Kotlin for Android development?

Interoperability

Compatible with Java, Java libraries, frameworks

You can call Kotlin code from Java & vice versa

New Features

Null safety, high-order functions & lambdas, smart casts, concise syntax, default & named arguments, type inference, data classes, sealed classes, coroutines

Tooling & IDE Support

Kotlin has excellent tooling support, especially with JetBrains' IntelliJ IDEA, Android Studio, and other major IDEs. This includes features like code completion, refactoring, debugging, and more

KAMI LIMU

Kotlin Features

Null Safety

Kotlin can distinguish between nullable & non-nullable types

```
var nonNullableString: String = "Hello"
var nullableString: String? = "Hello"
nullableString = null  // This is allowed
nonNullableString = null  // This causes a compile error
```

Extension Functions

Kotlin allows you to extend the functionality of classes without modifying their source code

```
fun String.addExclamation() = "$this!"
println("Hello".addExclamation())  // Output: Hello!
```

High-order Functions | Lambdas

High-order functions take other functions as parameters or return functions.

Lambda functions are anonymous functions often used as arguments to high-order functions. They provide a concise way of defining behavior in line without needing a full function.

A lambda in Kotlin is defined using `{}` and can have parameters followed by the `->` symbol and a function body

```
val sum: (Int, Int) -> Int = { x, y -> x + y }

fun main() {
    println(sum(3, 5))  // Output: 8
}
```

KAMI LIMU

Kotlin Features

Data Classes

Designed to hold data. They generate useful methods:

equals(), hashCode(), toString(), and copy()

```
data class User(val name: String, val age: Int)
val user = User("Alice", 25)
```

Type Inference

A powerful type inference system that reduces the need for explicit type declarations, making the code cleaner and more concise

```
val message = "Hello, Kotlin!"  // Type inferred as String
```

Default & Named Arguments

You can specify default values for function parameters and call functions using named arguments, increasing the readability of function calls

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

greet()  // Output: Hello, Guest!
greet("Alice")  // Output: Hello, Alice!
```

Kotlin Features

Coroutines

Built-in support for coroutines, to support asynchronous programming more straightforwardly and efficiently compared to traditional callback-based or thread-based approaches

```
import kotlinx.coroutines.*

fun main() = runBlocking {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
}
```

They are lightweight and can be suspended and resumed, making it easier to manage long-running tasks without blocking the main thread

Kotlin Features

Coroutines - Suspended Functions

Suspended functions allow you to write asynchronous code in a sequential style

A suspended function can be paused and resumed at a later time, making it ideal for tasks involving delays, such as network requests, file I/O, or complex computations

In the example, `delay()` is used to simulate a long-running operation i.e. network request

```
import kotlinx.coroutines.*

// Define a suspending function
suspend fun fetchData(): String {
    delay(1000L)  // Simulate a long-running task
    return "Data fetched"
}

fun main() = runBlocking {
    println("Start")

    // Launch a coroutine
    launch {
        val result = fetchData()
        println("Result: $result")
    }

    println("Doing other work...")
    delay(1500L)  // Simulate other work
    println("End")
}
```

KAMI LIMU

Unit Testing

What is unit testing?

This is a type of software testing where individual components of a software application are tested in isolation

Key Points

Purpose

To validate that each unit of the software performs as expected

Isolation

Units are tested in isolation from the rest of the application to ensure that any failures are due to issues within the unit itself, rather than dependencies or interactions with other parts of the system

Unit Testing

Automation

Unit tests are often automated to allow for frequent and quick execution

Automated tests can be run after every code change to catch regressions early and ensure the continued correctness of the software

Frameworks

Unit testing frameworks are implemented in different programming languages e.g JUnit

They provide tools for defining, executing, and organising tests

Assertions

Unit tests typically use assertions to verify that the actual output of a unit matches the expected behavior defined by the test case

If the assertion fails, it indicates a potential issue in the unit

Unit Testing

Mocking & Stubs

Dependencies relied upon by units are usually replaced with mocks or stubs to simulate their behaviour

Benefits

- Early bug detection
- Easy refactoring
- Improved code quality
- Promotes code reuse
- Regression testing
- Reduces debugging time
- Supports Continuous Integration/ Continuous Delivery (CI/CD)

KAMI LIMU

Unit Tests - Example

Testing functionality in a calculator.

Create test methods for each of the calculator methods

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

    fun subtract(a: Int, b: Int): Int {
        return a - b
    }
}
```

It is common to name the test class with the same name as the class it’s testing, but not required

To perform a unit test, call a method and then test the result against the expected output

```
import org.junit.jupiter.api.Assertions.assertEquals
import org.junit.jupiter.api.Test

class CalculatorTest {

    private val calculator = Calculator()

    @Test
    fun testAdd() {
        val result = calculator.add(2, 3)
        assertEquals(5, result)
    }

    @Test
    fun testSubtract() {
        val result = calculator.subtract(5, 3)
        assertEquals(2, result)
    }
}
```

Unit Testing

What are annotations?

Metadata added to test methods to provide instructions to the testing framework

Core Annotations (JUnit)

@Test - Marks a function as a test unit

@Before - A method run before each test

@After - A method run after each test

@BeforeClass - A method run once before any of the tests in the class

@AfterClass - A method run once after all tests in the class

@Ignore - Ignores the test method

@ParameterizedTest - Marks a method to be run with different parameters

Unit Testing

What are assertions?

Statements used to check if a certain condition is true

The core mechanism for validating the behaviour and output of the code being tested

If an assertion fails it indicates there is a problem with the unit

Common Assertions (JUnit)

assertEquals(expected, actual) - Checks if two values are equal

```
assertEquals(5, result, "2 + 3 should equal 5")
```

assertNotEquals(expected, actual) - Checks if two values are not equal

```
assertNotEquals(0, list.size())
```

Unit Testing

Common Assertions (JUnit)

assertTrue(condition) - Checks if a condition is true

```
assertTrue(result > 0)
```

assertFalse(condition) - Checks if a condition is false

```
assertFalse(list.isEmpty())
```

assertNull(object) - Checks if an object is null

```
val error: String? = null
assertNull(error)
```

Unit Testing

assertNotNull(object) - Checks if an object is not null

```
val error: String? = "An error occurred"
assertNotNull(error)
```

assertSame(expected, actual) - Checks that two objects do not refer to the same object instance

```
assertSame(singletonInstance, Singleton.getInstance())
```

assertNotSame(expected, actual) - Checks if two references point to different objects

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

val user1 = User("Alice", 30)
val user2 = User("Bob", 40)

assertNotSame(user1, user2)
```

Unit Testing

assertArrayEquals(expectedArray, actualArray) - Verifies that two arrays are equal

```
assertArrayEquals(arrayOf(1, 2, 3, 4), arrayOf(1, 2, 3, 4))
```

assertIterableEquals(expected, actual) - Checks that two iterables contain the same elements in the same order

```
assertIterableEquals(listOf(1, 2, 3, 4), listOf(1, 2, 3, 4))
```

assertEquals(expected, actual) - Checks if two values are equal

```
assertEquals(5, result, "2 + 3 should equal 5")
```

Unit Testing

Best Practices

- Write small focused tests
- Use descriptive test names
- Test positive & negative outcomes
- Make use of mocks & stubs
- Ensure tests are repeatable & consistent
- Test as much of your codebase as possible

Thank you.
Any Question?