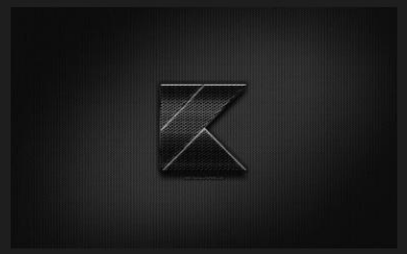


Kotlin & Compose Mastery

From History to Modern Android Development

A Comprehensive Guide

Agenda

- **Part 1:** History & Origins
- **Part 2:** Why Kotlin?
- **Part 3:** Syntax & Basics
- **Part 4:** Intermediate OOP
- **Part 5:** Functional Programming
- **Part 6:** Coroutines Intro
- **Part 7:** Jetpack Compose Basics
- **Part 8:** Advanced UI & State

Part 1: History

The Evolution of a Modern Language

The Origin Story

2011

Created by **JetBrains**. Unveiled as a new language for the JVM.

2016

Kotlin 1.0 Released. The first official stable version.

2017

Google announces **first-class support** for Kotlin on Android at Google I/O.

Part 2: Why Kotlin?

Modern features for modern developers

Conciseness

Java (Verbose)

```
public class Person { private String name; public Person(String name) { this.name = name; } public String getName() { return this.name; } }
```

Kotlin (Concise)

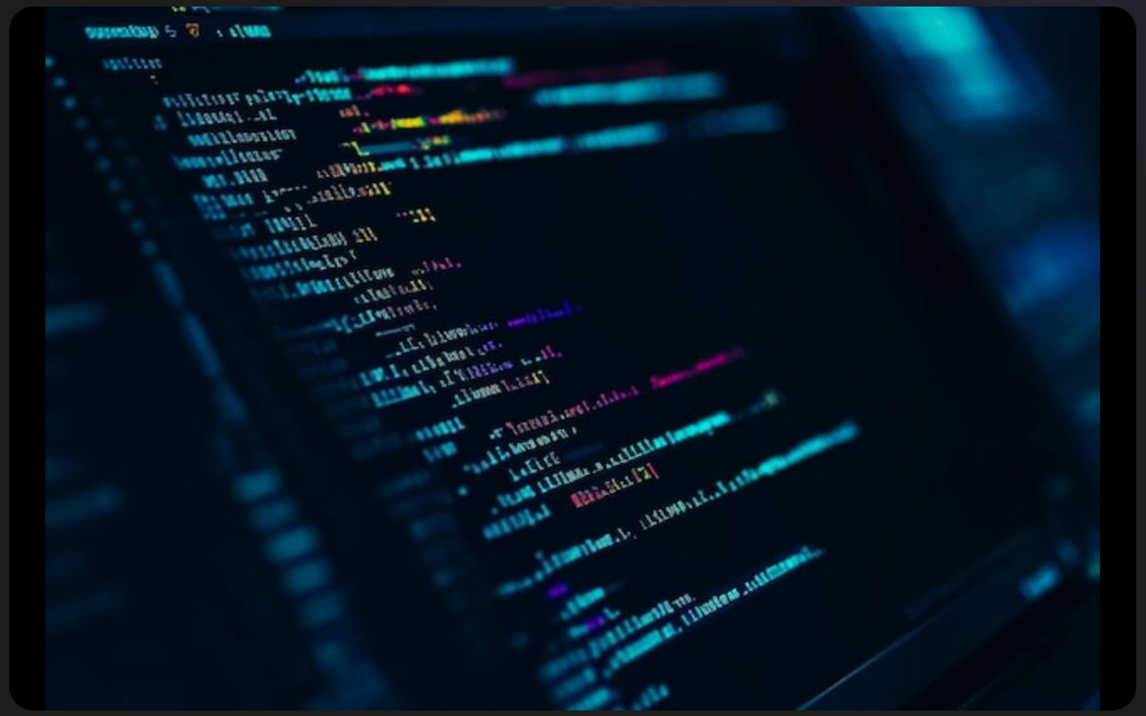
```
data class Person(val name: String) // That's it. Getters, setters, // equals(), hashCode() included.
```

Safety First

Null Safety

Kotlin's type system aims to eliminate the danger of null references, also known as `NullPointerException`.

- Compile-time checks
- Explicit nullable types
- Safe call operators



100% Interoperable

Seamless Mix

You can use Java and Kotlin files in the same project without any issues.

Libraries

Call Java libraries from Kotlin and vice versa. No need to rewrite existing code.

Part 3: Syntax & Basics

Building blocks of the language

Entry Point

The Main Function

An application entry point is a function named `main`.

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

Variables: Read-Only

Using 'val'

Always prefer `val` . It declares a read-only variable (immutable reference).

Value cannot be reassigned.

```
val name = "Kotlin" // name = "Java" //
Error! val pi = 3.1415
```

Variables: Mutable

Using 'var'

Use `var` when the value needs to change.

Value can be reassigned.

```
var score = 10 score = 20 // OK var isActive
= true isActive = false // OK
```

Type Inference

Kotlin compiler is smart. It infers the type from the initializer.

```
val language = "French" // Inferred as String val year = 2024 // Inferred as Int // Explicit type declaration is possible but optional val precise: Double = 3.14
```

Basic Types

Type	Examples	Description
Numbers	Byte, Short, Int, Long, Float, Double	Standard numeric primitives (wrapped as objects)
Boolean	true, false	Logic values
Char	'A', 'z', '\n'	Single 16-bit Unicode character
String	"Hello", "123"	Sequence of characters

String Templates

Concatenation is Old School

Use `$` to insert variables directly into strings.

Use `${}` for expressions.

```
val users = 10 println("Users: $users") val
price = 9.99 println("Total: ${price * 2}")
```

Functions

Structure

Defined with `fun` keyword.

Return type goes after the parameter list.

```
fun sum(a: Int, b: Int): Int { return a + b }
// Single-Expression body fun multiply(a:
Int, b: Int) = a * b
```

Named & Default Arguments

No more telescoping constructors or ambiguous parameters.

```
fun greet( name: String, prefix: String = "Hello" ) { ... } // Usage greet("John") // Uses default prefix
greet(prefix = "Hi", name = "Jane") // Named args
```

'if' is an Expression

Returns a Value

In Kotlin, `if` can return a value. It replaces the ternary operator `? :`.

```
val max = if (a > b) { a } else { b } // Or on one line val min = if (a < b) a else b
```

'when' Expression

Switch on Steroids

Replaces `switch`. Can check values, types, ranges, or arbitrary conditions.

```
when (x) { 1 → print("x is 1") 2 → print("x is 2") in 3..10 → print("x is 3-10") is String → print("x is String") else → print("unknown") }
```

Loops

```
// Iterate over a range for (i in 1..5) { println(i) } // Iterate over a collection for (item in items) {
println(item) } // While loop while (x > 0) { x-- }
```

The Billion Dollar Mistake

In Java, accessing a member of a null reference results in a `NullPointerException`, crashing the app.

```
String s = null; int length = s.length(); // Crash!
```

Nullable Types

Standard Types

By default, types cannot be null.

```
var a: String = "abc" // a = null // Compile Error
```

Nullable Types (?)

Add `?` to allow nulls.

```
var b: String? = "abc" b = null // OK
```

Safe Calls (?.)

Safely access properties of a nullable variable.

```
val b: String? = null val length = b?.length // Result: length is null (no crash) // If b was not null, it would return the length.
```

Elvis Operator (?:)

Provide a default value if the expression is null.

```
val l = b?.length ?: -1 // If b is null, l becomes -1 // Looks like Elvis's hair: ?:
```

Not-null Assertion (!!)

Forcing a nullable type to be non-null. Use with caution!

```
val l = b!!.length // If b is null, this throws NPE. // It's for when you are 100% sure it's not null.
```

Part 4: Intermediate OOP

Classes, Objects, and Inheritance

Classes

```
class Person( val firstName: String, var age: Int ) { // Initializer block init { println("Created $firstName")
} fun speak() { ... } }
```

Data Classes

Purpose

Classes solely to hold data. Automatically generates `toString`, `equals`, `hashCode`, `copy`.

```
data class User( val id: Int, val name:
String ) val u1 = User(1, "Alice") val u2 =
u1.copy(name = "Bob")
```

Inheritance

Classes are `final` by default. Use `open` to allow inheritance.

```
open class Shape { open fun draw() { ... } } class Circle : Shape() { override fun draw() { ... } }
```

Extension Functions

Add functionality to existing classes without inheriting from them.

```
fun String.addExclamation(): String { return this + "!" } val str = "Hello" println(str.addExclamation()) //
"Hello!"
```

Higher-Order Functions

Functions that take functions as parameters or return them.

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

Lambdas

Anonymous functions often passed to higher-order functions.

```
val sum = { x: Int, y: Int → x + y } // Trailing lambda syntax items.filter { it > 0 }
```

Collections

List

Ordered collection.

`listOf()`

Set

Unique elements.

`setOf()`

Map

Key-Value pairs.

`mapOf()`

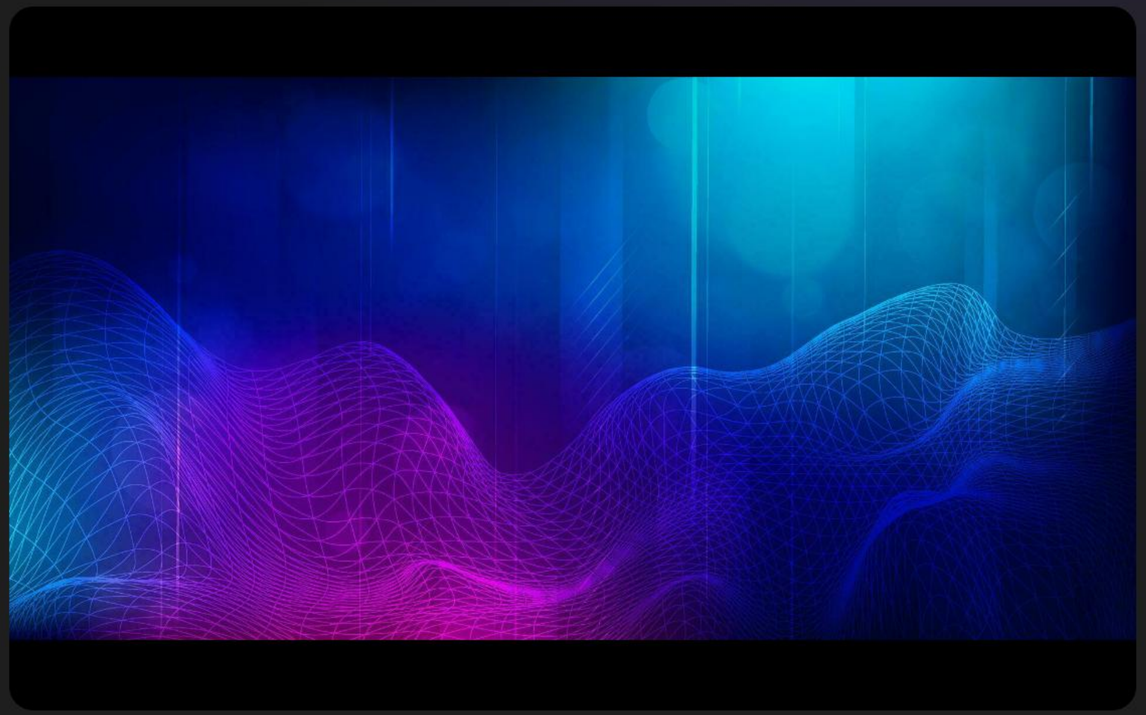
Functional Operations

```
val numbers = listOf(1, 2, 3, 4, 5) val doubled = numbers .filter { it % 2 == 0 } // Keep evens .map { it * 2 }
// Double them // Result: [4, 8]
```

Coroutines

Lightweight threads for asynchronous programming.

- Use `suspend` functions
- Non-blocking
- Structured concurrency



Suspend Function

```
suspend fun fetchUser(): User { delay(1000) // Non-blocking delay return User(...) } // Called inside a
CoroutineScope scope.launch { val user = fetchUser() updateUI(user) }
```

Thank You!

Questions?

Happy Coding