SE 3242: Android Application Development

Week 1: Kotlin Essentials

Session 1: From Dart to Kotlin

Engr. Daniel MOUNE
moune.daniel@ictuniversity.edu.cm
ICT University, Cameroon Campus, Yaoundé
P.O. Box 526, 1 Avenue Dispensaire Messassi, Zoatoupsi

Imparting ICTs in all academic disciplines

February 24, 2026

Today's Roadmap

Part 1: Why Kotlin? (15 min)

- Android's language of choice
- Kotlin vs. Dart: First impressions

Part 2: Variables & Basic Types (30 min)

- val vs var
- Type inference

Part 3: Null Safety (45 min)

- The billion-dollar mistake
- Nullable types, safe calls, Elvis

BREAK (15 min)

Part 4: Live Coding & Exercises (60 min)

- From Dart to Kotlin translation
- Pair programming

Part 5: Project Milestone (10 min)

"If you get lost, raise your hand – TAs are circulating."

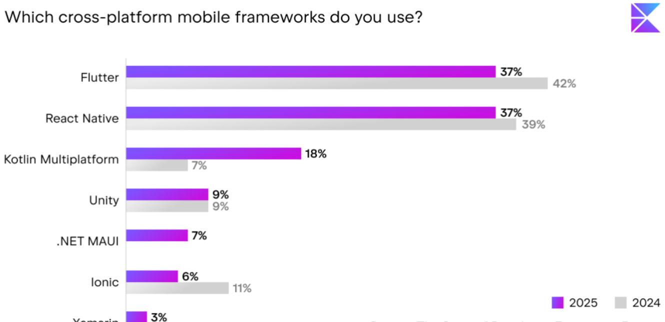
Why Kotlin? The Android Story

A Brief History

- 2017: Google announces first-class support
- 2019: Kotlin becomes preferred language for Android
- Today: 95% of top Android apps use Kotlin

Why Kotlin, not Java?

- More concise (40% less code than Java)
- Null safety built-in (fewer crashes)
- Interoperable with Java
- Functional + Object-Oriented



Kotlin vs Dart – First Glance

Dart

```
void main() {
  var name = 'Alice';
  print('Hello,␣$name');
}
```

Kotlin

```
fun main() {
  val name = "Alice"
  println("Hello,␣$name")
}
```

Observations

- Both use curly braces {}
- Kotlin uses `fun` not `function` or `void`
- String templates: both use $ but Kotlin no quotes inside template
- **IMPORTANT: Kotlin uses double quotes ONLY!**
  - `'hello'` → ✗ Error
  - `"hello"` → ✓ Correct

Variables – `val` (read-only) vs `var` (mutable)

THE GOLDEN RULE

Use `val` by default. Only use `var` when you MUST.

`val` (immutable reference)

```
val name = "Bob"
name = "Alice"   // ERROR
```

`var` (mutable reference)

```
var age = 25
age = 26            // OK Works
```

Dart Analogy

- Dart `final` → Kotlin `val`
- Dart `var` → Kotlin `var`

⚠ Nuance

- `val` means the **reference** can't change.
- The **object** itself can still be mutable.

Example:

```
val list = mutableListOf(1,2,3)
list.add(4)     // OK object changes
list = newList() // ERROR reference change
```

Type Inference - Kotlin Knows What You Mean

Inferred Types

```
val name = "Alice"          // String
var age = 25                // Int
val pi = 3.14159            // Double
val isRaining = false
// Boolean
```

You don't have to write the type - Kotlin infers it.

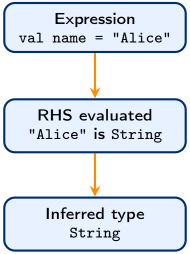
Explicit Types (Optional)

```
val name: String = "Alice"
var age: Int = 25
```

Watch Out!

Inference works from the right-hand side.

```
val x
// ERROR: need initializer
val y: Int
// OK, but must initialize later
```



Compiler uses RHS to determine type

Basic Types - Numbers, Text, and Logic

Integers

```
val a: Int = 100
val b: Long = 100L
val c: Byte = 127
val d: Short = 32767
```

Default Int for whole numbers.

Floating Point

```
val e: Double = 3.14
val f: Float = 3.14F
```

Double is default; use F suffix for Float.

Other Basics

```
val flag: Boolean = true
val letter: Char = 'A'
val text: String = "Kotl
```

Operations

```
val sum = 5 + 3              // 8
val isEqual = (5 == 5)       // true
val result = "Hello, " + "World"   // Concatenation
```

Type Conversion - You Must Ask Explicitly

⚠ Kotlin is strict about type conversion

No automatic widening like Java/C++.

```
val intNum = 42
val longNum: Long = intNum    // ERROR: Type mismatch
```

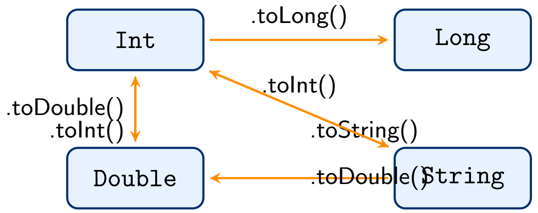
Use conversion functions

```
val intNum = 42
val longNum = intNum.toLong()
val doubleNum = intNum.toDouble()
val stringNum = intNum.toString()
```

Conversions for different types

```
"123".toInt()          // 123
"3.14".toDouble()      // 3.14
'a'.toInt()            // 97 (ASCII)
```

Type Conversion - Visual Overview



Conversion functions explicitly requested

Null Safety – Why Kotlin Eliminates NullPointerException

The Problem: Null References

- Tony Hoare invented null references in 1965 and later called it his **“billion-dollar mistake”**.
- In many languages, any reference can be `null`, leading to crashes.
- `NullPointerException` is a top cause of app crashes.

Dart/Java style (unsafe)

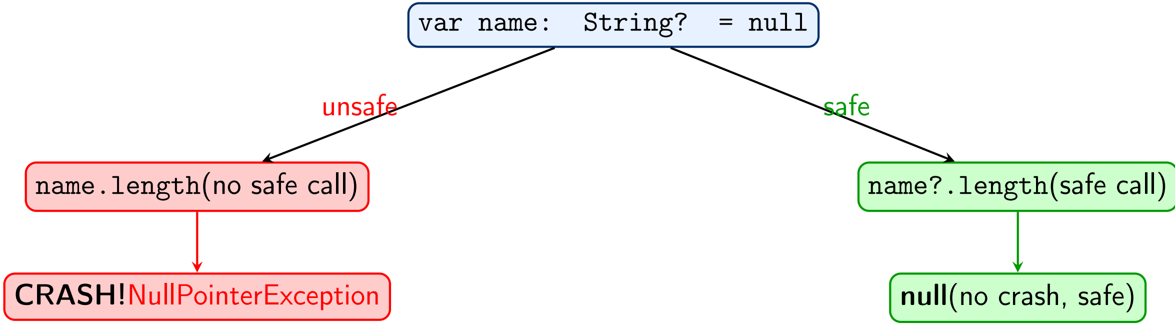
```
String name = null;
int len = name.length(); // CRASH!
```

Kotlin’s approach

```
var name: String? = null
val len = name?.length
// Safe: returns null
```

`?` marks nullable type; `?.` safe call.

Null Pointer Crash vs Safe Call Flow



Kotlin's type system
forces you to handle null

Working with Nullable Types

Declaring Nullable Variables

Add `?` after the type.

```
var neverNull: String = "hello"
neverNull = null    // ERROR

var canBeNull: String? = "hello"
canBeNull = null    // OK
```

Safe Call Operator ?.

Calls the method only if variable is not null.

```
val len = canBeNull?.length
// len is of type Int? (nullable Int)
```

Chaining Safe Calls

If any is null, result is null.

```
data class Address(val city: String?)
data class Person(val address: Address

val city = person?.address?.city
```

Elvis Operator `?:` – "If null, use this instead"

Problem: Provide a default value for a nullable expression

```
val name: String? = getUserName()
val displayName = if (name != null) name else "Guest"
```

This works, but Kotlin offers a concise alternative.

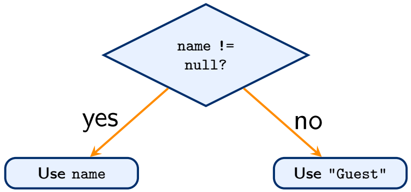
Elvis Operator `?:`

```
val displayName = name ?: "Guest"
```

If `name` is not null, use its value; otherwise use `"Guest"`.

"Elvis: Because the King never leaves you null."

Elvis Operator – Visual Flow & Safe Call Chaining



Chaining with Safe Calls

```
val city = person?.address?.city ?: "Unknown"
```

If any step is null, the whole chain yields null, then Elvis gives default.

Not-null Assertion `!!` – "I know this isn't null!"

⚠ DANGER ZONE

`!!` converts any nullable type to a non-null type, but throws `NullPointerException` if the value is null.

```
val name: String? = maybeNull()
val length = name!!.length    // CRASH if name is null
```

When is it acceptable?

- You are **absolutely certain** the value is not null.
- In tests or quick prototypes (but refactor later).
- When interacting with Java code that lacks nullability annotations.

Better alternatives

- Use safe calls `?.` and Elvis `?:`
- Use `if`-check with smart cast
- Use `requireNotNull()` or `checkNotNull()` for clearer intent

`if` is an Expression, Not Just a Statement

In Kotlin, `if` returns a value

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

No ternary operator (`? :`) needed - `if` does it all.

Multi-line `if`

```
val grade = if (score >= 90) {
    println("Excellent!")
    "A"
} else if (score >= 80) {
    "B"
} else {
    "C"
}
```

Comparison with Dart

Dart: `var max = (a > b) ? a : b;`

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

`Last expression in each block is the returned value.`

"Remember: if is an expression – it produces a value!"

`when` – The Swiss Army Knife of Conditionals

Replaces `switch` from other languages

```
when (x) {
    1 -> println("x␣==␣1")
    2, 3 -> println("x␣==␣2␣or␣3")
    in 4..10 -> println("x␣in␣range␣4..10")
    else -> println("x␣is␣something␣else")
}
```

`when` as expression

```
val description = when (x) {
    1 -> "one"
    2 -> "two"
    else -> "many"
}
```

Without argument (boolean)

```
when {
    x.isOdd() -> "odd"
    x.isEven() -> "even"
    else -> "impossible"
}
```

"`when` is more powerful than Dart's `switch` – it handles any condition."

Now It's Your Turn! Live Coding & Pair Programming

What We've Covered So Far

- Variables: `val` vs `var`
- Basic types and type inference
- Null safety: `?`, `?.`, `?:`, `!!`
- Conditionals: `if` and `when`

Live Coding Session (60 min)

We'll translate a Dart program to Kotlin together, step by step.

1. Open Android Studio / Kotlin Playground
2. We'll write a simple "Student Grade Calculator"
3. Use `data class`, nullable inputs, `when` for grades
4. Handle edge cases with Elvis and safe calls

Pair Programming & Project Milestone

Pair Programming

After live coding, you'll work in pairs on similar exercises. TAs will be circulating. Don't hesitate to ask!

Project Milestone 1

Create a Kotlin data class for your app idea and submit by end of week.
Requirements:

- Define a data class with at least 3-4 properties
- Use appropriate types (`String`, `Int`, `Double`, `Boolean`)
- Use nullable types where it makes sense
- Include documentation comments

Live Coding: Student Grade Calculator

Problem Statement

Write a program that:

- Takes a student's name and score (could be null/missing).
- If score is null, print "No score for [name]".
- Otherwise, determine the grade using:
  - 90-100: A
  - 80-89: B
  - 70-79: C
  - 60-69: D
  - Below 60: F
- Print "[name] scored [score] : Grade [grade]".

Live Coding: Steps 1 & 2

Step 1 – Define a data class

```
data class Student(val name: String, val score: Int?)
```

Step 2 – Create a function to get grade

```
fun getGrade(score: Int): Char = when (score) {
    in 90..100 -> 'A'
    in 80..89  -> 'B'
    in 70..79  -> 'C'
    in 60..69  -> 'D'
    else       -> 'F'
}
```

Live Coding: Handling Null Scores

Step 3 – Process a list of students

```
fun printGrades(students: List<Student>) {
    for (student in students) {
        val gradeMessage = student.score?.let { score ->
            val grade = getGrade(score)
            "${student.name}␣scored␣$score␣:␣Grade␣$grade"
        } ?: "No␣score␣for␣${student.name}"

        println(gradeMessage)
    }
}
```

`?.let` executes block only if score not null, then Elvis provides default.

Live Coding: Testing with Sample Data

Step 4 – Test with sample data

```
fun main() {
    val students = listOf(
        Student("Alice", 95),
        Student("Bob", 82),
        Student("Charlie", null),
        Student("Diana", 47)
    )
    printGrades(students)
}
```

Exercise 1: Null-Safe Data Processing

Task

You have a list of `User` objects with nullable email addresses.

```
data class User(val name: String, val email: String?)
val users = listOf(
    User("Alex", "alex@example.com"),
    User("Blake", null),
    User("Casey", "casey@work.com")
)
```

Requirements

1. Print the email addresses for users who have them, in uppercase.
2. For users without email, print "[Name] has no email".
3. Count how many users have valid emails and print the total.

Exercise 2: Temperature Descriptions

Task

Write a function `describeTemperature(temp: Int?) : String` that returns:

- "Freezing" if temp ≤ 0
- "Cold" if 1–15
- "Mild" if 16–25
- "Warm" if 26–35
- "Hot" if 36–45
- "Extreme" if >45
- "No data" if temp is null

Bonus

Create a list of temperatures (some null) and print descriptions using a loop. Use a `when` expression (without argument) to handle ranges.

Exercise 3: Filtering and Transforming Collections

Task

Given a list of nullable integers:

```
val numbers: List<Int?> = listOf(1, null, 3, null, 5, 6, null, 8)
```

Perform these operations

1. Filter out nulls, keep only non-null values.
2. Double each remaining number.
3. Sum the doubled values.
4. Print the sum using `filterNotNull()`, `map()`, and `sum()`.

One-liner challenge

Can you do it in a single chain?

Project Milestone 1: Your App's Data Model

Objective

Create a Kotlin `data class` that represents the core data of your app idea.

Requirements

- Choose an app idea (from your proposal or brainstorm now).
- Define one or more `data classes` with appropriate properties.
- Use nullable types where data might be missing.
- In a `main` function, create at least 3 instances and print them.
- Include comments explaining your choices (why certain properties are nullable, etc.).

Project Milestone 1: Submission

Submission Details

Due: **Next Monday 23:59**

Submit a single `.kt` file on the LMS.

File name: `Milestone1_YourName.kt`

This is the foundation of your final project – make it thoughtful!