{ this is Kotlin }

Introducing Kotlin

Tiberiu Tofan

What is Kotlin

- Modern
- General purpose language
- Statically typed
- OOP
- FP

What is Kotlin

- Safe
- Concise
- Expressive
- Interoperable with Java
- Multiplatform
- Industrial targeted

What is Kotlin

- Developed by JetBrains
- Open Source
- KEEP - Kotlin Evolution and Enhancement Process

A bit of history

2011 - JetBrains announces Project Kotlin

2012 - Open sourced

2016 - v1.0 released

2017 - Google announces first-class support for Kotlin in Android

- v1.2 - experimental multiplatform

2018 - v1.3 - coroutines, contracts

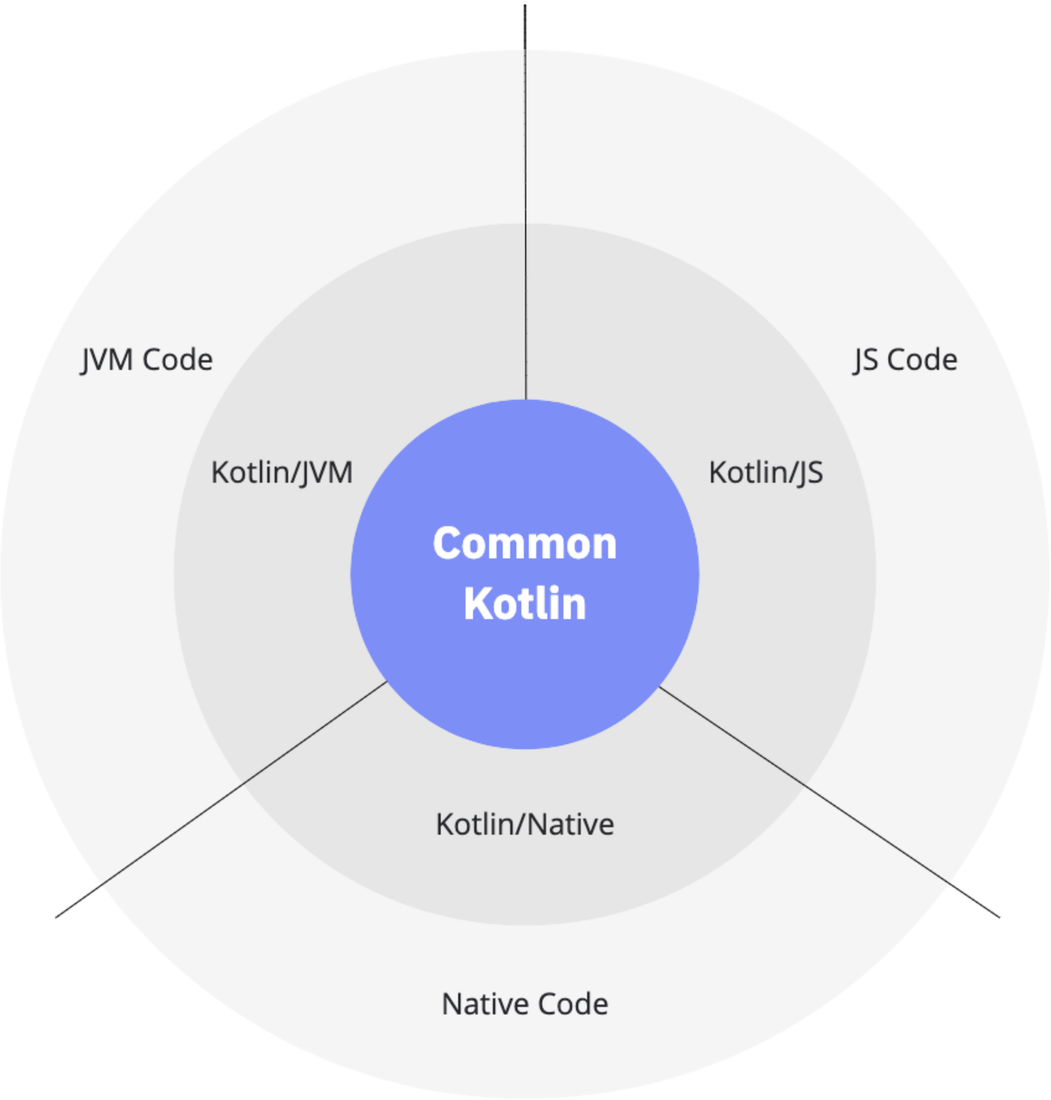
2019 - Preferred language for Android

2020 - v1.4 - better Objective-C/Swift interop

2021 - v1.5 - JVM IR compiler

Kotlin Multiplatform

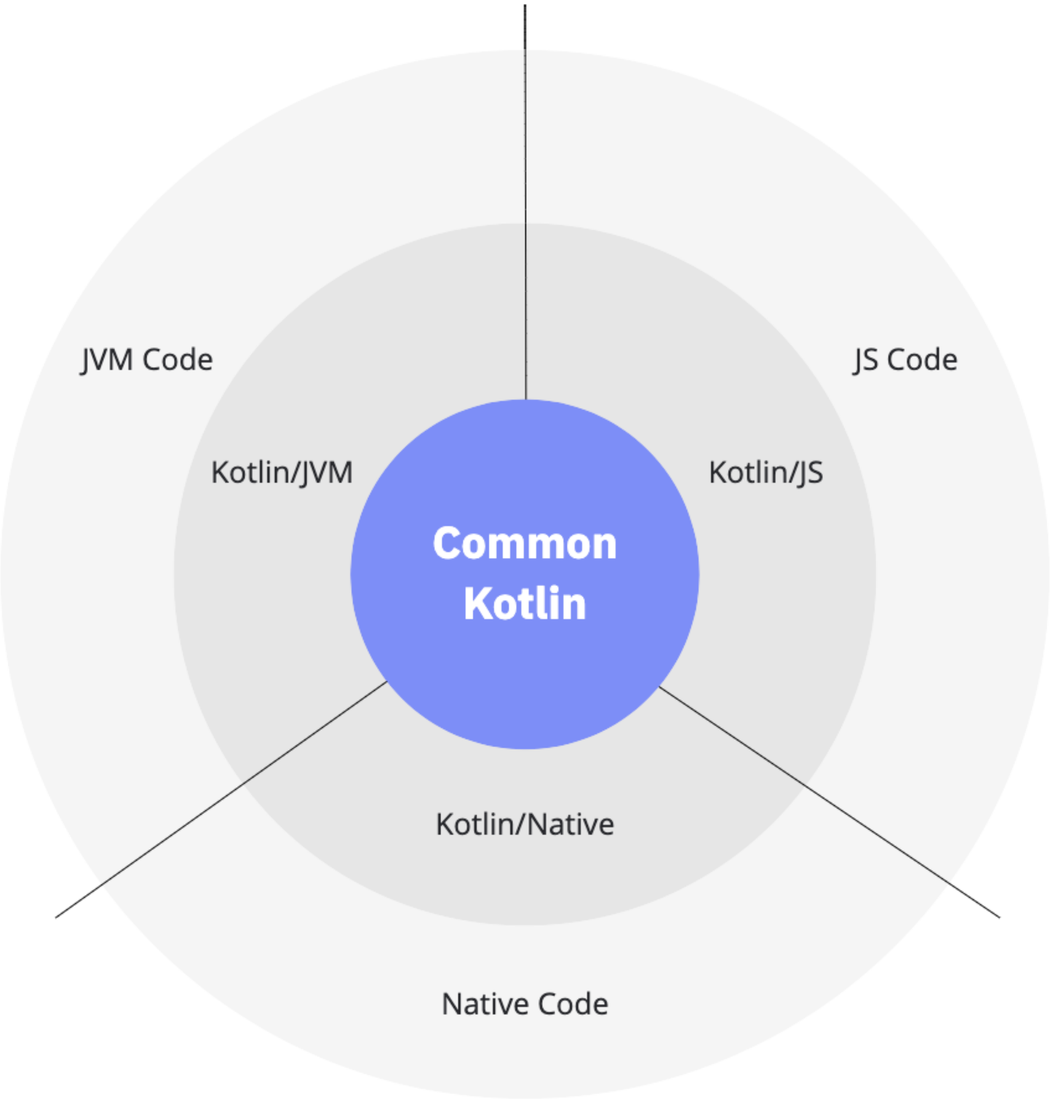
- Kotlin/JVM
  - Server side
  - Android
- Kotlin/JS
- Kotlin/Native
  - macOS
  - iOS
  - Android NDK
  - Linux
  - Windows



source: kotlinlang.org

Kotlin Multiplatform

- **Kotlin/JVM**
  - **Server side**
  - Android
- Kotlin/JS
- Kotlin/Native
  - macOS
  - iOS
  - Android NDK
  - Linux
  - Windows



source: kotlinlang.org

Hello, World!

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

```
fun main(args: Array<String>)
```

Variables

- **val** - read-only - corresponds to final in java

```
val language: String = "Kotlin"
language = "Java"
```

compilation error: val cannot be reassigned

- **var** - mutable

```
var version: Double = 1.5
version = 1.6
```

OK

❗ prefer **vals** to **vars**

Variables

nullability

```
val language: String = null
```

compilation error: Null can not be a value of a non-null type String

```
val language: String? = null
```

OK

```
var version: Double? = 1.5
version = null
```

OK

❗ prefer **non-null** types

Type Inference

```
: String
val title = "Type Inference"

: Int
var slideNumber = 10

slideNumber = "Slide 10"
```

compilation error: Type mismatch: inferred type is String but Double was expected

- the type of the variable can be omitted
- the compiler will infer the type automatically
- Kotlin is statically typed: every variable has a type
- the type of the variable is the same type as the type of the expression on the left of the assignment

! omit the type only when it is clear from context

Type Inference

```
: Nothing?
val x = null
```

Expression vs Statement

Statement

- a sequence of instructions
- does not evaluate to a value
- usually used for side effects
- standalone expressions (evaluated value not used) can be considered statements

Expression

- combination of variables, constants, operators and functions
- evaluates to a single value
- always has a type

Statement

Expression

```
var x = 2
```

Statement

Expression

```
x += 3
```

```
var x = 2
```

Statement

Expression

```
if (temperature > 20) "hot" else "cold"
```

```
var x = 2
x += 3
```

Statement

Expression

```
if (temperature > 20) println("hot")
```

```
var x = 2
x += 3
```

```
if (temperature > 20) "hot" else "cold"
```

Statement

Expression

```
while(temperature > 25) {
    runCoolingCycle()
}
```

```
var x = 2
x += 3
if (temperature > 20) println("hot")
```

```
if (temperature > 20) "hot" else "cold"
```

Statement

```
when (grade) {
    5 -> "sufficient"
    in (6..7) -> "satisfactory"
    8 -> "good"
    9 -> "very good"
    10 -> "excellent"
    else -> "failed"
}
```

Expression

```
var x = 2
x += 3
if (temperature > 20) println("hot")
while(temperature > 25) {
    runCoolingCycle()
}
```

```
if (temperature > 20) "hot" else "cold"
```

Statement

Expression

```
temperature + 5
```

```
var x = 2
x += 3
if (temperature > 20) println("hot")
while(temperature > 25) {
    runCoolingCycle()
}
```

```
if (temperature > 20) "hot" else "cold"
when (grade) {
    5 -> "sufficient"
    in (6..7) -> "satisfactory"
    8 -> "good"
    9 -> "very good"
    10 -> "excellent"
    else -> "failed"
}
```

Statement

Expression

```
try { "42".toInt() }
catch (e: NumberFormatException) { 0 }
```

```
var x = 2
x += 3
if (temperature > 20) println("hot")
while(temperature > 25) {
    runCoolingCycle()
}
```

```
if (temperature > 20) "hot" else "cold"
when (grade) {
    5 -> "sufficient"
    in (6..7) -> "satisfactory"
    8 -> "good"
    9 -> "very good"
    10 -> "excellent"
    else -> "failed"
}
temperature + 5
```

Statement

```
var x = 2
x += 3
if (temperature > 20) println("hot")
while(temperature > 25) {
    runCoolingCycle()
}
```

Expression

```
if (temperature > 20) "hot" else "cold"
when (grade) {
    5 -> "sufficient"
    in (6..7) -> "satisfactory"
    8 -> "good"
    9 -> "very good"
    10 -> "excellent"
    else -> "failed"
}
temperature + 5
try { "42".toInt() }
catch (e: NumberFormatException) { 0 }
```

Statement

Expression

```
val eval =
```

```
if (temperature > 20) "hot" else "cold"

when (grade) {
    5 -> "sufficient"
    in (6..7) -> "satisfactory"
    8 -> "good"
    9 -> "very good"
    10 -> "excellent"
    else -> "failed"
}

temperature + 5

try { "42".toInt() }
catch (e: NumberFormatException) { 0 }
```

Functions

Basics

- declaration

```
fun duplicate(s: String): String {
    return s + s
}
```

- type inference

```
fun duplicate(s: String) {
    return s + s
}
```

: String

- function with expression body

```
fun duplicate(s: String) = s + s
```

Functions

Unit return type

```
: Unit
fun sayHello() {
    println("Hello, World!")
}
```

- A function body with no explicit return has *Unit* as the default return type
- Unit is usually inferred, not specified explicitly
- Indicates side effects

Functions

Default & named parameters

- default parameter value

```
fun duplicate(s: String, separator: String = "") = s + separator + s
println(duplicate("Kotlin"))
```

KotlinKotlin

- named parameters

: String

```
println(duplicate("Kotlin", separator = "_"))
```

Kotlin_Kotlin

- alternative implementation: string interpolation

```
fun duplicate(s: String, separator: String = "") = "$s$separator$s"
println("duplicated: ${duplicate("Kotlin")}")
```

duplicated: KotlinKotlin

Functions

Context of function declarations

- Top-level functions

```
package dev.school

fun topLevelFunction(): String = "top level"
```

- Member functions

```
class MyClass {
    fun memberFunction(): String = "member function"
}
```

- Local functions

```
fun topLevelFunction(): String {
    fun localFunction() = "local function"
    return localFunction()
}
```

Conditionals

if

- if is an expression

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

- branches of if can be blocks

```
val max = if (a > b) {
    println("a is greater than b")
    a
} else {
    println("b is greater or equal to a")
    b
}
```

❗ avoid if as a statement

Conditionals

when

```
val ects = when (grade) {
    in (5..6) -> "E"
    7 -> "D"
    8 -> "C"
    9 -> "B"
    10 -> "A"
    else -> "F"
}
val result = when {
    grade >= 5 -> "passed"
    else -> {
        println("reschedule exam")
        "failed"
    }
}
```

- conditional expression with multiple branches
- similar to *switch* in other languages
- branches can be blocks of code
- evaluates sequentially each branch until one is true
- exhaustive - all possible cases must be covered
- can be used as a statement

Loops
(do-)while

```
while(condition) {
    /*
     */
}

do {
    /*
     */
} while (condition)
```

Loops

for

```
val ns = listOf(299, 792, 458)

for (n in ns) {
    print(n)
}
```

```
299792458
```

```
for ((i, n) in ns.withIndex()) {
    println("the element at $i is $n")
}
```

```
the element at 0 is 299
the element at 1 is 792
the element at 2 is 458
```

Loops

for over maps

```
val designers = mapOf(
    "Scala" to "Martin Odersky",
    "Java" to "James Gosling",
    "Kotlin" to "JetBrains",
    "Groovy" to "James Strachan",
    "Closure" to "Rich Hickey")

for ((language, designer) in designers) {
    println("$language is designed by $designer")
}
```

```
Scala is designed by Martin Odersky
Java is designed by James Gosling
Kotlin is designed by JetBrains
Groovy is designed by James Strachan
Closure is designed by Rich Hickey
```

Classes

declaration and default constructor

```
class Account(val iban: String, val product: String, val currency: String, val balance: Double)
val account = Account("NL69INGB0123456789", "Current Account", "EUR", 100.0)
```

- *iban*, *product*, etc. are properties
- class definition acts as primary constructor
- constructor is invoked as a function, no `new` keyword required
- all other function properties apply to constructors: named arguments, default arguments, etc.

Classes

initialization

```
class Account(val iban: String, val product: String, val currency: String, val balance: Double) {
    init {
        require(currency.length == 3) { "Currency should be a 3 chars code" }
    }
}
```

- The primary constructor cannot contain any code
- Initialization code can be placed in *initializer blocks*, which are prefixed with the `init` keyword
- require is a standard library function that throws `IllegalArgumentException` if the condition is not met

Exceptions

- throwing exceptions

```
throw Exception("Please retry!")
```

- catching exceptions

```
try {
    // some code
} catch (e: Exception) {
    // handler
} finally {
    // optional finally block
}
```

- `try` is an expression

```
val n = try { "42".toInt() } catch (e: NumberFormatException) { 0 }
```