

Mobile Computing

Writing Kotlin for Android

CC BY-SA 4.0, T. Amberg, FHNW; Based on
CC BY 2.5, Android Open Source Project
Slides: tmb.gr/mc-ktn



Overview

These slides introduce programming in Kotlin.

Basic concepts of the Kotlin language.

How they compare to Java.

Slides are new. Found an issue? [Let me know.](#)

Prerequisites

Access the [Kotlin Playground online editor](#).

Edit the code there to print your name.

Run the code to see the result.

Kotlin language

"Kotlin is a multiplatform, statically typed, general-purpose programming language."

"[Inspired by] Java, Scala, C# and Groovy. [Intended to be] pragmatic, i.e., being a programming language useful for day-to-day development [...]"

— [Kotlin Language Specification](#)

Compared to Java

Kotlin compiles to JVM bytecode, among others.

Many language **concepts are similar** to Java.

There seems to be less clutter in Kotlin.

Kotlin **can call Java** and **vice versa**.

Kotlin for Android

The default is to **develop Android apps with Kotlin**.

Android Studio has full tooling support for Kotlin.

Android committed to a "**Kotlin-first approach**".

Libraries can be **used from / written in** Kotlin.

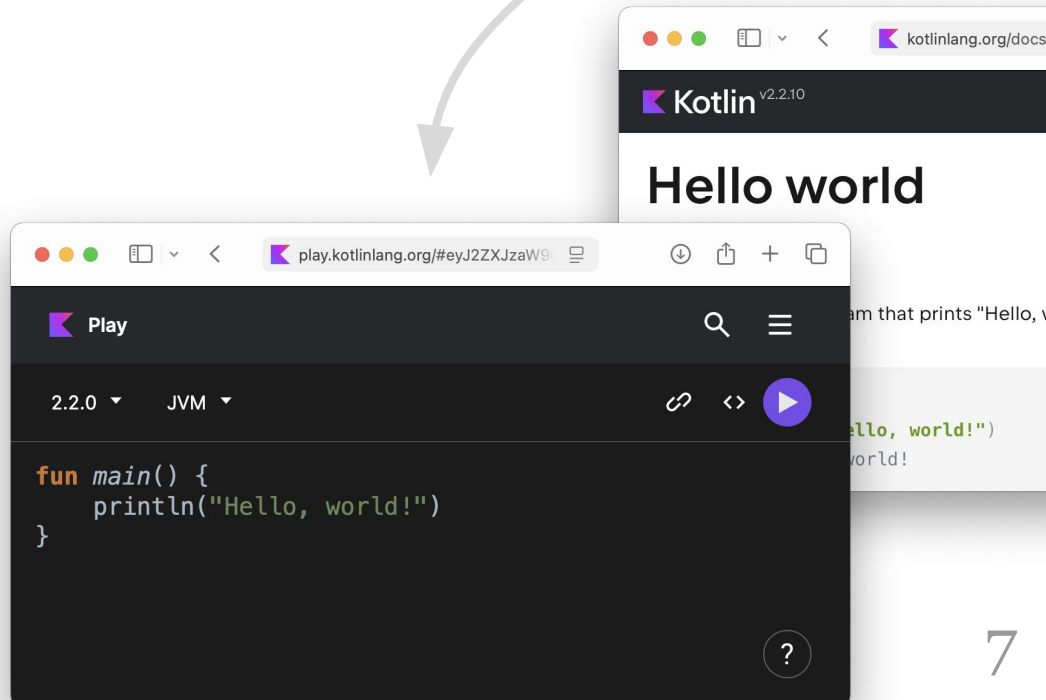
Hello, world!

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

.kt shows this code in
play.kotlinlang.org

.html links to docs on
kotlinlang.org/docs

[.kt](#) | [.html](#)



Variables

[.kt](#) | [.html](#)*

Declare read-only values with *val* (recommended).

```
val i = 3 // read-only  
i = 2 // error: 'val' cannot be reassigned
```

Declare mutable variables with *var* (if needed).

```
var x = 5 // mutable  
x = 7
```

*CTRL-click to open links in a new tab.

Basic types

[.kt](#) | [.html](#)

A type (here *Int*) can either be inferred, or explicit.

```
val i = 3 // inferred type, initialized
```

```
val j: Int // explicit type, required for  
j = 4 // late initialization of value
```

```
val k: Int = 5 // explicit type, redundant
```

Basic types are *Byte*, *Short*, *Int*, *Long*, *UByte*, *UShort*, *UInt*, *ULong*, *Float*, *Double*, *Boolean*, *Char*, *String*.

String templates

[.kt](#) | [.html](#)

String templates are useful to reference variables.

```
val what = "hi"  
val whom = "folks"  
val text = "$what, $whom!"  
val stat = "(${text.length} chars)"
```

Concatenation with + is possible, but less concise.

```
val copy = what + ", " + whom + "!"
```

Arrays

[.kt](#) | [.html](#)

Array<*T*> is a Kotlin class (vs. part of the language).

```
val pets = arrayOf("dog", "cat", "snake")  
val legs: Array<Int?> = arrayOfNulls(3)  
legs[0] = 4;
```

To keep primitive types unboxed, use *IntArray*, etc.

```
val legs = IntArray(3) // Java: = new int[3];
```

If performance matters, use Arrays, else Collections. 11

Collections

[.png](#) | [.html](#)

The Kotlin library contains three collection types.

Lists — ordered collections of (duplicate) items.

Sets — unordered collections of unique items.

Maps — sets of keys that map to one value each.

Each of these have read-only and mutable versions.

Note: Arrays are not Collections, always mutable.

Lists

[.kt](#) | [.html](#)

Lists of type *List*<*E*> are read-only.

```
val list = listOf("hello", "hola", "hi")  
val item = list[0] // or .get(0)
```

Lists of type *MutableList*<*E*> can be modified.

```
val mutableList = list.toMutableList()  
mutableList[0] = "ciao" // or .set(0, "...")
```

Sets

[.kt](#) | [.html](#)

Sets of type *Set*<*T*> are read-only, items are unique.

```
val animalList = listOf("cow", "cow", "horse")  
val speciesSet = animalList.toSet()
```

Sets of type *MutableSet*<*T*> can be modified.

```
val mutableSet = speciesSet.toMutableSet()  
mutableSet.remove("cow") // or -=  
mutableSet.add("cattle") // or +=
```

Maps

[.kt](#) | [.html](#)

Maps of type *Maps* $\langle K, V \rangle$ are read-only, keys are a Set.

```
val dataMap = mapOf("temp" to 23, "humi" to 42)
val t = dataMap["temp"] // or .get("temp")
val keySet = dataMap.keys
```

Maps of type *MutableMap* $\langle K, V \rangle$ can be modified.

```
val mutableMap = dataMap.toMutableMap()
mutableMap["temp"] = 5 // or .put("temp", 5)
```

Conditionals

[.kt](#) | [.html](#)

Kotlin provides *if* and *when* to check conditions.

Both can be used as a statement, or an expression.

If statements (like *if* in Java), must use curly braces.

```
if (cond) { ... } else { ... }
```

If expressions (like the *? :* ternary operator in Java).

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


When statement

[.kt](#) | [.html](#)

When statement (*switch* in Java)

```
when (ch) {  
    'a' -> result = 'A' // break  
    else -> result = '?' // default  
}
```

When expression

```
val result = when (ch) { 'a' -> 'A' }
```

Ranges

[.kt](#) | [.html](#)

Range *to* or *until* the last element, with optional *step*.

```
val r = 1..4 // 1, 2, 3, 4 or 1.rangeTo(4)
val r2 = 1..<4 // 1, 2, 3 or 1.rangeUntil(4)
val r3 = 1..10 step 2 // or .step(2)
```

Ranges work for other types, like *Double* and *Char*.

```
val r4 = 1.0..4.0 // according to IEEE-754
val r5 = 'a'..'d' // 'a', 'b', 'c', 'd'
```

Loops

[.kt](#) | [.html](#)

For iterates over an *Iterable*, e.g. a range or collection.

```
for (i in 1..3) { println(i) }  
for (item in list) { println(item) }
```

While and *do ... while* a condition holds (as in Java).

```
var i = 0  
while (i < 4) { i++ }  
do { i-- } while (i > 0)
```

Functions

[.kt](#) | [.html](#)

Declaring a function with parameters, return type.

```
fun sum(x: Int, y: Int): Int { return x + y }  
fun log(x: Double, b: Double = 2.0): Double ...
```

Calling the function with (named, default) arguments.

```
val i = sum(3, 5)  
val d = log(x = 100.0, b = 10.0) // named args  
val e = log(128.0) // default argument b = 2.0
```

Classes

[.kt](#) | [.html](#)

Declare a *class* with *properties* in its header or body.

```
class Contact (val id: Int, var email: String)
class Project (val p1: Contact) { val team ... }
```

Add *member functions* (methods in Java) to its body.

```
class Project (...) { // class name, constructor
    fun start() { ... } // member function
}
```

Instances

[.kt](#) | [.html](#)

Create an *instance* by calling a *constructor* of a class.

```
val c = Contact(23, "me@example.com")  
val p = Project(c) // no new keyword
```

Access *properties* and call *member functions*.

```
c.email = "you@example.com"  
println(p.team.size)  
p.start()
```

Constructors

[.kt](#) | [.html](#)

The *constructor* keyword allows visibility modifiers.

```
class A() // public class, public constructor  
class B private constructor() // non-public c.
```

Secondary constructor, calling the primary one, *this()*.

```
class C(val i: Int, val j: Int) { // primary c.  
    constructor(i: Int): this(i, 0) // 2ndry c.  
}
```

Visibility

[.kt](#) | [.html](#)

Visibility modifiers include *public*, which is the default.

The *internal* modifier reduces visibility to the module*.

And *private* reduces it to the containing file or class.

```
val a = 1 // public, visible from everywhere
internal val b = 2 // ... *unit of compilation
private val c = 3 // ... inside same .kt file
class C () { private val d = 4 } // ... class
```


Null safety

[.kt](#) | [.html](#)

Values can be *null* in Kotlin, but not by default.

```
var s: String = null // cannot be a value
```

A type like String becomes *nullable* by adding ?

```
var s2: String? = null
```

Safe calls ?. return *null* if an instance is null.

```
val n = s2?.length
```

Hands-on, 15': Kotlin basics

Extend the [example](#) on p. 22 with additional features.

- Allow the *email* property of a contact to be *null*.
- Hide the *team* property inside the *Project* class.
- Add a new member function *.addContact(...)*.
- Limit the number of contacts per team to 5.
- Make sure each contact is only added once.

Discuss your solutions with your peers.

Lambda expressions

[.kt](#) | [.html](#)

A function can be written as a lambda expression.

```
fun len(s: String): Int { return s.length }  
val len2 = { s: String -> s.length } // lambda
```

A lambda expression can be called like a function.

```
val m = len("hola") // calling a function  
val n = len2("hey") // calling a lambda
```

Filters

[.kt](#) | [.html](#)

A filter takes a lambda expression as a predicate.

```
val range = 1..10 // or array, collection, etc.  
val isEven = { i: Int -> i % 2 == 0 }
```

Calling *filter()* checks the predicate for each item.

```
val evens = range.filter(isEven)  
val odds = range.filter({ i -> i % 2 == 1 })
```

For another typical use of lambdas, see *map()*.

Extension functions

[.kt](#) | [.html](#)

Add a new function, here *len()*, to an existing class.

```
fun String.len(): Int { return this.length }  
// or fun String.len(): Int = this.length  
// or fun String.len() = this.length  
  
val s = "hello" // type String, not our class  
val n = s.len() // extended member function
```

Extension functions are dispatched statically.

Class inheritance

[.kt](#) | [.html](#)

A class can be a subclass of another (non-final) class.

```
open class B // default w/o open would be final  
class C: B() // c.tor calls superclass c.tor
```

The top level parent class is *Any* (like *object* in Java).

```
val c = C()  
val isC = c is B // true  
val isAny = c is Any // true
```

Abstract classes

[.kt](#) | [.html](#)

An *abstract* (base) class can be inherited by default.

```
abstract class B { // not instantiated
    abstract fun m() // no implementation
} // can have non-abstract members/functions
```

Subclasses implement abstract members/functions.

```
class C: B() { override fun m() { ... } }
```

The *override* modifier prevents ambiguity.

Interfaces

[.kt](#) | [.html](#)

A class can implement one or more interfaces.

```
interface I { // not instantiated, no c.tor  
    fun m() // no impl., abstract by default  
} // only abstract members/functions
```

Subclasses implement members/functions.

```
class C() : I { override fun m() { ... } }
```

It's possible to delegate interface implementation.

Objects

[.kt](#) | [.html](#)

The keyword *object* allows to declare a singleton.

```
object C { fun m() { ... } } // no constructor
```

To call such a single instance, use the object name.

```
C.m() // single instance, not an extension
```

Inheriting from classes works, so do interfaces.

```
object D : B(), I, J { ... }
```

Companion objects

[.kt](#) | [.html](#)

An object inside a class can be declared a *companion*.

```
class C private constructor() {  
    companion object Factory {  
        fun create(): C = C()  
    }  
}
```

Call companion object members via the class name.

```
val c = C.create() // like static in Java
```

Generics

[.kt](#) | [.html](#)

Write type safe code without relying on specific types.

```
class Hat<T>() { var item: T? = null }  
class Bunny()  
class Dove() // our item types
```

The T in $Hat<T>$ is set (to *Bunny*) at compile time.

```
val h = Hat<Bunny>() // hat for bunnies  
h.item = Bunny() // instance assigned  
h.item = Dove() // type mismatch
```

Exceptions

[.kt](#) | [.html](#)

Exceptions allow to *throw* and *catch* runtime errors.

```
throw MyException() // unchecked (unlike Java)
```

require(), *check()*, *error()* functions throw exceptions.

```
check(condition) { "error message" } // throws
```

The *try-catch-finally* blocks work the same as in Java.

```
try { ... } catch (e: MyE...) { ... } finally { ... }
```

Null safety revisited

[.kt](#) | [.html](#)

Smart casts (implicit) and safe casts, using *A as? B*.

```
val b: B = a as? B // or null if not A is B
```

Early returns *x?*, combined with the Elvis operator *?:*:

```
var n = users[userId]?.friends?.size ?: -1  
n = list.sumOf { (e as? String)?.length ?: 0 }
```

If any part is null, evaluation stops, "returns" null.

Hands-on, 15': Intermediate topics

Refactor the [example](#) on p. 22 and add features.

- Create two subclasses of *Contact*, *Dev* and *Suit*.
- Add a way to set a dev's programming language.
- Prevent contacts from being instantiated directly.
- Extend *Project* to filter the team by devs or suits.
- Give* suits a "manager", devs a ["maker" schedule](#).

*A property *schedule* of type *String* is good enough.

Libraries and APIs

[.kt](#) | [.html](#)

Libraries distribute reusable code for common tasks.

The Kotlin [standard library](#) is imported by default.

```
println("nice"); // part of kotlin.* library
```

Other packages, classes, etc. have to be imported.

```
import kotlin.time.* // or .Duration, etc.
```

```
val pause: Duration = 15.minutes
```

Summary

We saw the basics of programming in Kotlin.

It's similar to Java, with a more concise syntax.

Combining proven concepts and an elegant library.

Next: Getting Started with Android.

Challenge: Advanced topics

Study these concepts, we might meet them later on*.

- **Special classes** like *data*, *enum* and *value* classes.
- **Properties** *fields*, delegated *by*, *lazy* initialisation.
- **Generics** *in*, *out*, *where*, co- and contravariance.
- **Coroutines**, for asynchronous, concurrent tasks.
- **Scope functions** *let*, *apply*, *run*, *also*, and *with*.

*In existing code, without implementation aspects.

Feedback or questions?

Write me on Teams or email

thomas.amberg@fhnw.ch

Thanks for your time.