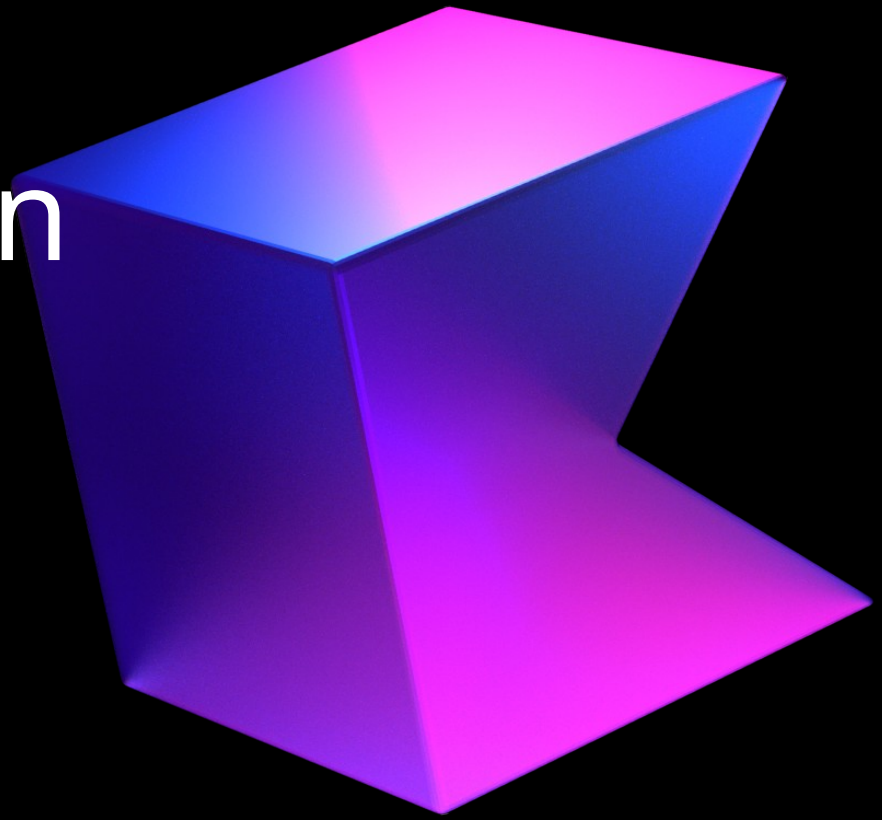
JET BRAINS Kotlin

Introduction to Kotlin

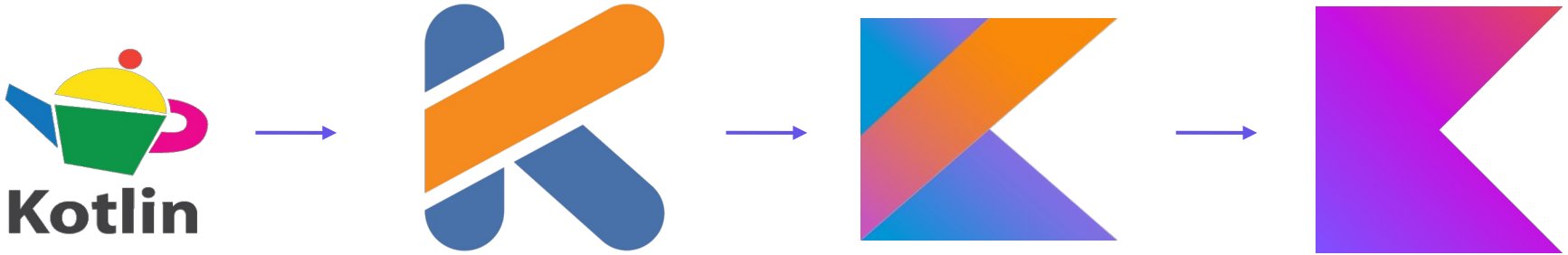


@kotlin | Developed by JetBrains

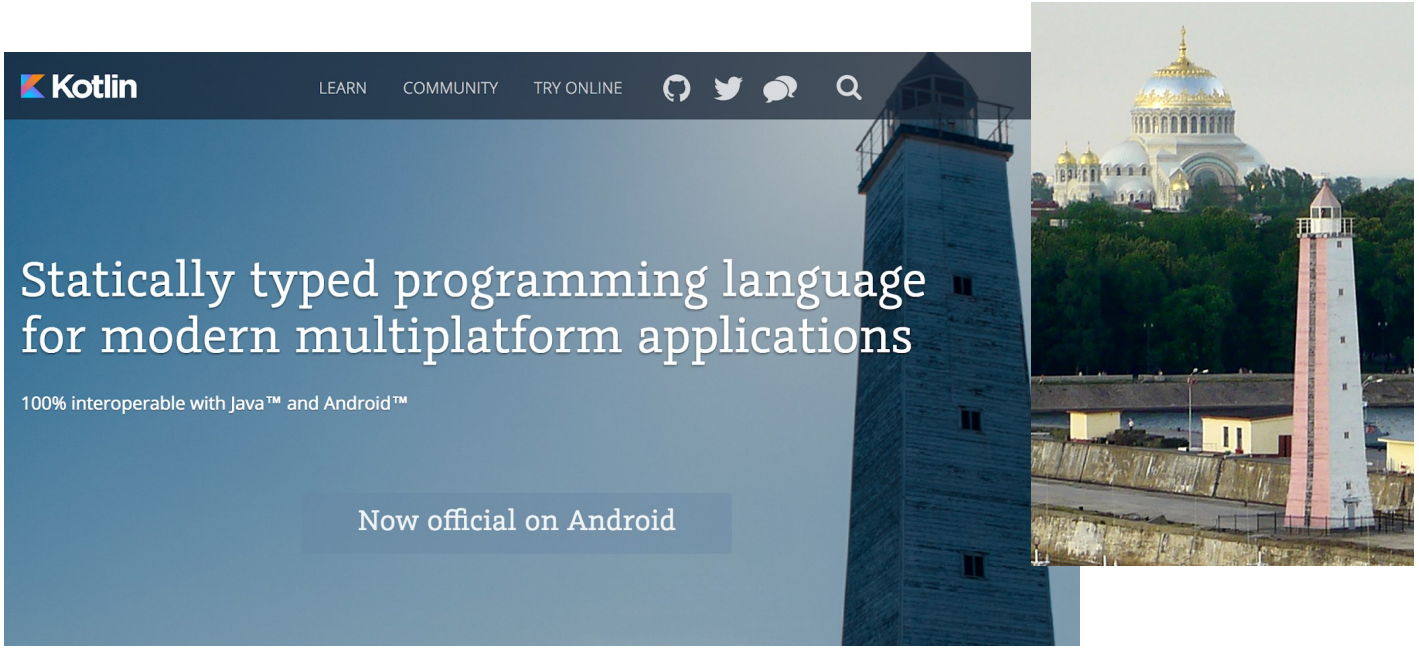
Why Kotlin?

- Expressiveness/Conciseness
- Safety
- Portability/Compatibility
- Convenience
- High Quality IDE Support
- Community
- Android 👀
- More than a gazillion devices run ~~Java~~ Kotlin
- Lactose free
- ~~Sugar free~~
- Gluten free

Logo



Name



Kotlin is named after an island in the Gulf of Finland.

Hello, world!

```
fun main(args: Array<String>) {
        println("Hello, world!")
}

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

fun main() = println("Hello, world!")
```

Where is “;”???

The basics

```
fun main(args: Array<String>) {
        print("Hello")
        println(", world!")
}
```

- An entry point of a Kotlin application is the main **top-level** function.
- It accepts a variable number of String arguments that can be omitted.
- print prints its argument to the standard output.
- println prints its arguments and adds a line break.

Variables

val/var myValue: Type = someValue

- var - mutable
- val - immutable
- Type can be inferred in most cases
- Assignment can be deferred

```
val a: Int = 1   // immediate assignment

var b = 2                    // 'Int' type is inferred
b = a                                // Reassigning to 'var' is okay

val c: Int                   // Type required when no initializer is provided
c = 3                                // Deferred assignment
a = 4                                // Error: Val cannot be reassigned
```

Variables

const val/val myValue: Type = someValue

- const val - compile-time const value
- val - immutable value
- for const val use uppercase for naming

```
const val NAME = "Kotlin"          // can be calculated at compile-time
val nameLowered = NAME.lowercase()          // cannot be calculated at compile-time
```

Functions

```
fun sum(a: Int, b: Int): Int {
                return a + b
}

fun mul(a: Int, b: Int) = a * b

fun printMul(a: Int, b: Int): Unit {
                println(mul(a, b))
}

fun printMul1(a: Int = 1, b: Int) {
                println(mul(a, b))
}

fun printMul2(a: Int, b: Int = 1) = println(mul(a, b))
```

Single expression function.

Unit means that the function does not return anything meaningful.

It can be omitted.

Arguments can have **default** values.

If expression

```
fun maxOf(a: Int, b: Int): Int {
        if (a > b) {
                return a
        } else {
                return b
        }
}
```


```
fun maxOf(a: Int, b: Int) =
        if (a > b) {
                a
        } else {
                b
        }
```

if can be an expression (it can return).

Can be a one-liner:

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

When expression

```
when (x) {
    1 -> print("x == 1")
    2 -> print("x == 2")
    else -> {
        print("x is neither 1 nor 2")
    }
}
```

when returns, the same way that if does.

```
when {
    x < 0 -> print("x < 0")
    x > 0 -> print("x > 0")
    else -> {
        print("x == 0")
    }
}
```

The condition can be inside of the branches.

When statement

```
fun serveTeaTo(customer: Customer) {
                val teaSack = takeRandomTeaSack()

    when (teaSack) {
        is OolongSack -> error("We don't serve Chinese tea like $teaSack!")
        in trialTeaSacks, teaSackBoughtLastNight ->
                error("Are you insane?! We cannot serve uncertified tea!")
    }

    teaPackage.brew().serveTo(customer)
}
```

when can accept several options in one branch. else branch can be omitted if when block is used as a *statement*.

&& vs and

`if (a && b) { ... }` VS `if (a and b) { ... }`

Unlike the `&&` operator, this function does not perform short-circuit evaluation.

The same behavior with OR:

`if (a || b) { ... }` VS `if (a or b) { ... }`

Loops

```
val items = listOf("apple", "banana", "kiwifruit")

for (item in items) {
    println(item)
}

for (index in items.indices) {
    println("item at $index is ${items[index]}")
}

for ((index, item) in items.withIndex()) {
    println("item at $index is $item")
}
```

Loops

```
val items = listOf("apple", "banana", "kiwifruit")

var index = 0
while (index < items.size) {
    println("item at $index is ${items[index]}")
    index++
}

var toComplete: Boolean
do {
    ...
    toComplete = ...
} while(toComplete)
```

The condition variable can be initialized inside to the do...while loop.

Loops

There are break and continue labels for loops:

```
myLabel@ for (item in items) {
    for (anotherItem in otherItems) {
        if (...) break@myLabel
        else continue@myLabel
    }
}
```

Ranges

```
val x = 10
if (x in 1..10) {
    println("fits in range")
}

for (x in 1..5) {
    print(x)
}

for (x in 9 downTo 0 step 3) {
    print(x)
}
```

downTo and step are extension functions, not keywords.

'..' is actually T.rangeTo(that: T)

Null safety

```
val notNullText: String = "Definitely not null"
val nullableText1: String? = "Might be null"
val nullableText2: String? = null

fun funny(text: String?) {
        if (text != null)
                println(text)
        else
                println("Nothing to print :(")
}

fun funnier(text: String?) {
        val toPrint = text ?: "Nothing to print :("
        println(toPrint)
}
```

Elvis operator ?:

If the expression to the left of ?: is not null, the Elvis operator returns it; otherwise, it returns the expression to the right.

Note that the expression on the right-hand side is evaluated only if the left-hand side is null.

```
fun loadInfoById(id: String): String? {
        val item = findItem(id) ?: return null
        return item.loadInfo() ?: throw Exception("...")
}
```

Safe Calls

someThing?.otherThing does not throw an NPE if someThing is null.

Safe calls are useful in chains. For example, an employee may be assigned to a department (or not). That department may in turn have another employee as a department head, who may or may not have a name, which we want to print:

```
fun printDepartmentHead(employee: Employee) {
        println(employee.department?.head?.name)
}
```

To print only for non-null values, you can use the safe call operator together with let:

```
employee.department?.head?.name?.let { println(it) }
```

Unsafe Calls

The not-null assertion operator (!!) converts any value to a non-null type and throws an NPE exception if the value is null.

```
fun printDepartmentHead(employee: Employee) {
        println(employee.department!!.head!!.name!!)
}
```

Please, avoid using unsafe calls!

TODO

Always throws a NotImplementedError at **run-time** if called, stating that operation is not implemented.

```
// Throws an error at run-time if calls this function, but compiles
fun findItemOrNull(id: String): Item? = TODO("Find item $id")


// Does not compile at all
fun findItemOrNull(id: String): Item? = { }
```

String templates and the string builder

```
val i = 10
val s = "Kotlin"

println("i = $i")
println("Length of $s is ${s.length}")

val sb = StringBuilder()
sb.append("Hello")
sb.append(", world!")
println(sb.toString())
```

Lambda expressions

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

According to Kotlin convention, if the last parameter of a function is a function, then a lambda expression passed as the corresponding argument can be placed outside the parentheses:

```
val badProduct = items.fold(1, { acc, e -> acc * e })

val goodProduct = items.fold(1) { acc, e -> acc * e }
```

If the lambda is the only argument, the parentheses can be omitted entirely (the documentation calls this feature "trailing lambda as a parameter"):

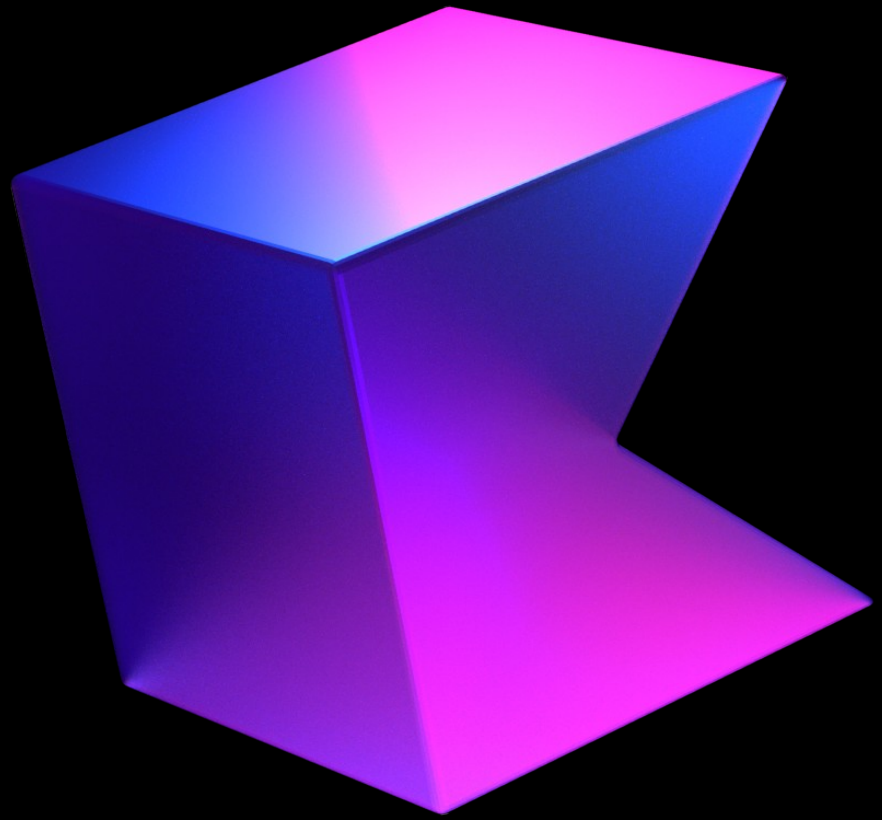
```
run({ println("Not Cool") })
run { println("Very Cool") }
```

When in doubt

Go to:

- kotlinlang.org
- kotlinlang.org/docs
- play.kotlinlang.org/byExample

Thanks!



@kotlin | Developed by JetBrains