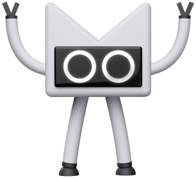
Kotlin - aimed to make developers happier

Maciej Procyk



November 15, 2021

Table of contents

Introduction

Say "Hello World" and go home

HelloWorld.kt

```
fun main() = println("Hello, Kotlin World")
```

Short history background

- unveiled in 2011 (being designed from 2010) by JetBrains

Short history background

- unveiled in 2011 (being designed from 2010) by JetBrains
- goal: compile as quickly as Java but offer as rich features as Scala

Short history background

- unveiled in 2011 (being designed from 2010) by JetBrains
- goal: compile as quickly as Java but offer as rich features as Scala
- you guess a name source (think of Java)

Short history background

- unveiled in 2011 (being designed from 2010) by JetBrains
- goal: compile as quickly as Java but offer as rich features as Scala
- named for the island Kotlin near Saint Petersburg

Short history background

- unveiled in 2011 (being designed from 2010) by JetBrains
- goal: compile as quickly as Java but offer as rich features as Scala
- named for the island Kotlin near Saint Petersburg
- announced as official Android language in 2017

Kotlin

Short history background

- unveiled in 2011 (being designed from 2010) by JetBrains
- goal: compile as quickly as Java but offer as rich features as Scala
- named for the island Kotlin near Saint Petersburg
- announced as official Android language in 2017
- becomes multiplatform programming language

Main design goals

- concise, object-oriented language

Kotlin

Main design goals

- concise, object-oriented language, ”better” than Java

Main design goals

- concise, object-oriented language, ”better” than Java but fully operable with it

Main design goals

- concise, object-oriented language, "better" than Java but fully operable with it
- enable fast turnaround for developer

Main design goals

- concise, object-oriented language, "better" than Java but fully operable with it
- enable fast turnaround for developer
- provide many ways to reuse your code

Main design goals

- concise, object-oriented language, "better" than Java but fully operable with it
- enable fast turnaround for developer
- provide many ways to reuse your code
- expressiveness - new syntax features

Main design goals

- concise, object-oriented language, "better" than Java but fully operable with it
- enable fast turnaround for developer
- provide many ways to reuse your code
- expressiveness - new syntax features
- scalability - introduce coroutines for JVM

Main design goals

- concise, object-oriented language, "better" than Java but fully operable with it
- enable fast turnaround for developer
- provide many ways to reuse your code
- expressiveness - new syntax features
- scalability - introduce coroutines for JVM
- enable systems migration step-by-step

Main design goals

- concise, object-oriented language, ”better” than Java but fully operable with it
- enable fast turnaround for developer
- provide many ways to reuse your code
- expressiveness - new syntax features
- scalability - introduce coroutines for JVM
- enable systems migration step-by-step
- provide great tooling and easy learning curve

Use cases

- main Android language

Use cases

- main Android language
- server-side development

Use cases

- main Android language
- server-side development
- multiplatform (mobile) programming

Use cases

- main Android language
- server-side development
- multiplatform (mobile) programming
- frontend development

Use cases

- main Android language
- server-side development
- multiplatform (mobile) programming
- frontend development
- being introduced to data science

Kotlin

Use cases

- main Android language
- server-side development
- multiplatform (mobile) programming
- frontend development
- being introduced to data science
- even tried in competitive programming

Language

"Simplify" our greeting

HelloWorldKt.kt

```
object HelloWorldKt {
  @JvmStatic
  fun main(args: Array<String>): Unit {
      val name: String? = readLine()
      if (name == null) return
      System.out.println("Hello " + name)
  }
}
```

Imperative approach

HelloWorld.kt

```
fun main() {
  val name = readLine() ?: return
  println("Hello $name")
}
```

Kotlin

Declarative approach

HelloWorld.kt

```
fun main() = readLine()
  ?.let { println("Hello $it") }
  ?: Unit
```

Basic language assumptions

- items are   by default

Basic language assumptions

- items are `public` by default

Basic language assumptions

- items are `public` by default
- classes and methods are by default

Basic language assumptions

- items are `public` by default
- classes and methods are `final` by default

Basic language assumptions

- items are `public` by default
- classes and methods are `final` by default
- there is no concept in mind

Basic language assumptions

- items are `public` by default
- classes and methods are `final` by default
- there is no `static` concept in mind

Kotlin

Basic language assumptions

- items are `public` by default
- classes and methods are `final` by default
- there is no `static` concept in mind (but the abstraction of `companion object` exists)

Basic language assumptions

- items are `public` by default
- classes and methods are `final` by default
- there is no `static` concept in mind (but the abstraction of `companion object` exists)

Basic language assumptions

- items are `public` by default
- classes and methods are `final` by default
- there is no `static` concept in mind (but the abstraction of `companion object` exists)
- emphasis of for every type

Basic language assumptions

- items are `public` by default
- classes and methods are `final` by default
- there is no `static` concept in mind (but the abstraction of `companion object` exists)
- emphasis of null-ability for every type with `?`

Basic language assumptions

- items are `public` by default
- classes and methods are `final` by default
- there is no `static` concept in mind (but the abstraction of `companion object` exists)
- emphasis of null-ability for every type with ?
- no primitive types

Kotlin

Basic language assumptions

- items are `public` by default
- classes and methods are `final` by default
- there is no `static` concept in mind (but the abstraction of `companion object` exists)
- emphasis of null-ability for every type with `?`
- no primitive types but quite a lot of optimizations including effective usage of them

Basic language assumptions

- items are `public` by default
- classes and methods are `final` by default
- there is no `static` concept in mind (but the abstraction of `companion object` exists)
- emphasis of null-ability for every type with `?`
- no primitive types but quite a lot of optimizations including effective usage of them
- `final` variables used by default

Basic language assumptions

- items are `public` by default
- classes and methods are `final` by default
- there is no `static` concept in mind (but the abstraction of `companion object` exists)
- emphasis of null-ability for every type with `?`
- no primitive types but quite a lot of optimizations including effective usage of them
- `final` variables used by default with the keyword `val`

Basic language assumptions

- items are `public` by default
- classes and methods are `final` by default
- there is no `static` concept in mind (but the abstraction of `companion object` exists)
- emphasis of null-ability for every type with `?`
- no primitive types but quite a lot of optimizations including effective usage of them
- `final` variables used by default with the keyword `val`
- high usage of type inference in different contexts (`val`s and `var`s used as fields and variables definitions)

Program structure

- functions as first-class citizens (functional types of type `(A, B) -> C`)

Program structure

- functions as first-class citizens (functional types of type `(A, B) -> C`)
- not only `classes` and `interfaces`, but also `objects` (and `sealed` versions of them) and `value classes`

Program structure

- functions as first-class citizens (functional types of type `(A, B) -> C`)
- not only `classes` and `interfaces`, but also `objects` (and `sealed` versions of them) and `value classes`
- no `package private` visibility but `public`, `protected`, `private` and `internal` are used

Program structure

- functions as first-class citizens (functional types of type `(A, B) -> C`)
- not only `classes` and `interfaces`, but also `objects` (and `sealed` versions of them) and `value classes`
- no `package private` visibility but `public`, `protected`, `private` and `internal` are used
- statically typed variables with:

Program structure

- functions as first-class citizens (functional types of type `(A, B) -> C`)
- not only `classes` and `interfaces`, but also `objects` (and `sealed` versions of them) and `value classes`
- no `package private` visibility but `public`, `protected`, `private` and `internal` are used
- statically typed variables with:
  - number types like `Byte`, `Short`, `Int`, `Long`, (with unsigned versions) `Float`, `Double` (all with explicit conversion)

Program structure

- functions as first-class citizens (functional types of type `(A, B) -> C`)
- not only `classes` and `interfaces`, but also `objects` (and `sealed` versions of them) and `value classes`
- no `package private` visibility but `public`, `protected`, `private` and `internal` are used
- statically typed variables with:
  - number types like `Byte`, `Short`, `Int`, `Long`, (with unsigned versions) `Float`, `Double` (all with explicit conversion)
  - `String`, `Char` and `Boolean` types

Program structure

- functions as first-class citizens (functional types of type `(A, B) -> C`)
- not only `classes` and `interfaces`, but also `objects` (and `sealed` versions of them) and `value classes`
- no `package private` visibility but `public`, `protected`, `private` and `internal` are used
- statically typed variables with:
  - number types like `Byte`, `Short`, `Int`, `Long`, (with unsigned versions) `Float`, `Double` (all with explicit conversion)
  - `String`, `Char` and `Boolean` types
  - `Array<T>` and specialized versions for primitives `IntArray`, `DoubleArray`, ...

Program structure

- functions as first-class citizens (functional types of type `(A, B) -> C`)
- not only `classes` and `interfaces`, but also `objects` (and `sealed` versions of them) and `value classes`
- no `package private` visibility but `public`, `protected`, `private` and `internal` are used
- statically typed variables with:
  - number types like `Byte`, `Short`, `Int`, `Long`, (with unsigned versions) `Float`, `Double` (all with explicit conversion)
  - `String`, `Char` and `Boolean` types
  - `Array<T>` and specialized versions for primitives `IntArray`, `DoubleArray`, ...
  - ends on `Any` and includes also `nullable` types

Kotlin

Null safety - elvis operator

ElvisOperator.kt

```kotlin
val beOrNotToBe: String? = "be"
```

Null safety - elvis operator

ElvisOperator.kt

```kotlin
val beOrNotToBe: String? = "be"

val answerSize: Int? = beOrNot?.length
```

Null safety - elvis operator

ElvisOperator.kt

```
val beOrNotToBe: String? = "be"
val answerSize: Int? = beOrNot?.length
val decided: String = beOrNot ?: "not"
```

Kotlin

Null safety - elvis operator

ElvisOperator.kt

```
val beOrNotToBe: String? = "be"

val answerSize: Int? = beOrNot?.length

val decided: String = beOrNot ?: "not"

val sureAnswerSize: Int = beOrNot!!.length
```

Kotlin

Null safety - elvis operator

ElvisOperator.kt

```
val beOrNotToBe: String? = "be"
val answerSize: Int? = beOrNot?.length
val decided: String = beOrNot ?: "not"
val sureAnswerSize: Int = beOrNot!!.length
```

also supported in safe casting with `as?` operator

Lambdas definitions

Kotlin introduces new syntax for lamdas i.e. { }

Lambdas.kt

```
val hasSense: (Int) -> Boolean = { it == 42 }
```

Lambdas definitions

Kotlin introduces new syntax for lamdas i.e. { }

Lambdas.kt

```
val hasSense: (Int) -> Boolean = { it == 42 }

val beep = { println("beep") }
```

Lambdas definitions

Kotlin introduces new syntax for lamdas i.e. { }

Lambdas.kt

```
val hasSense: (Int) -> Boolean = { it == 42 }

val beep = { println("beep") }

val printWarn = { msg: Any ->
  println("WARN: $msg")
}
```

Lambdas definitions

Kotlin introduces new syntax for lamdas i.e. { }

Lambdas.kt

```
val hasSense: (Int) -> Boolean = { it == 42 }

val beep = { println("beep") }

val printWarn = { msg: Any ->
  println("WARN: $msg")
}

beep(); printWarn(hasSense(42))
```

Expression controlled flow

FlowFeatures.kt

```
val state = if (readLine() == "42") "alive"
            else "dead"
```

Expression controlled flow

FlowFeatures.kt

```
val state = if (readLine() == "42") "alive"
            else "dead"

val mood =
  when (val grade = (2..6).random()) {
    in 3..4 -> "uff... I passed with $grade"
    5, 6 -> "happy getting $grade"
    else -> "sad because of $grade"
  }
```

Named loops and scopes

FlowFeatures.kt

```
val friends = listOf("Alice", "Bob", "Carol")

for (friend in friends)
  println("Hello $friend")

friends@ for (friend in friends) {
  for (i in 0..3) {
    println("Hello $friend")
    if (friend.last() == 'a') break@friends
  }
}
```

Fast return from flow

FlowFeatures.kt

```
while (true) {
  readLine()
    ?.let(::println)
    ?: break
}
```

Fast return from flow

FlowFeatures.kt

```
while (true) readLine()?.let(::println) ?: break
```

Fast return from flow

FlowFeatures.kt

```
while (true) {
  readLine()
    ?.let(::println)
    ?: break
}

val name = readLine() ?: return
val surname = readLine() ?: return
println("Your full name is $name $surname")
```

Kotlin

Functions in Kotlin

BasicFunctions.kt

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

fun sub(a: Int, b: Int) = a - b
```

Functions in Kotlin

BasicFunctions.kt

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

fun sub(a: Int, b: Int) = a - b

val op = ::add
```

Kotlin also supports functions references that belong to one of the `KFunction<out R>` subtypes

Default arguments

SyntaxFeaturesArgs.kt

```
fun fib(
  idx: UInt,
  curr: Long = 1,
  last: Long = 0,
): Long =
  if (idx == 0U) curr
  else fib(idx - 1U, curr + last, curr)

fib(idx = 42U, curr = 42, last = 1)
fib(42U)
```

Kotlin

Extension functions

ExtensionFunctions.kt

```
// this can be omitted
fun Any.printHash() =
  println(hashCode())

// also for final classes
fun String.twice() = this + this

"42".twice().printHash()
```

Defined as `static` functions with first hidden parameter desugared from `this`

Infix functions

InfixFunctions.kt

```
infix fun Int.times(i: Int) = this * i
val life = 6 times 7
```

Infix functions

InfixFunctions.kt

```
infix fun Int.times(i: Int) = this * i
val life = 6 times 7
```

StdLibInfixFunctions.kt

```
val numberName = mapOf(
  1 to "one",
  2 to "two",
)
```

Kotlin

Operators

Predefined set of operators in language like `+a`, `!a`, `a++`, `a..b`, `a + b`, `a in b` (and much more) that can be manually defined for every type

OperatorFunctions.kt

```
operator fun String.not() = this.map {
  if (it.isUpperCase()) it.lowercaseChar()
  else it.uppercaseChar()
}.joinToString("")
```

Operators

Predefined set of operators in language like `+a`, `!a`, `a++`, `a..b`, `a + b`, `a in b` (and much more) that can be manually defined for every type

OperatorFunctions.kt

```
operator fun String.not() = this.map {
  if (it.isUpperCase()) it.lowercaseChar()
  else it.uppercaseChar()
}.joinToString("")

println(!"wOw")
```

and used as standard operators.

High order functions

HighOrderFunctions.kt

```
fun <T, U> Iterable<T>.foldLeft(
  initAcc: U, f: (U, T) -> U
): U {
  var acc = initAcc
  for (element in this) acc = f(acc, element)
  return acc
}
```

Idiomatic Kotlin code uses convention of having function parameters as last ones.

High order functions

HighOrderFunctions.kt

```
fun <T, U> Iterable<T>.foldLeft(
  initAcc: U, f: (U, T) -> U
): U {
   var acc = initAcc
   for (element in this) acc = f(acc, element)
   return acc
}

numbers.foldLeft(0) { acc, e -> acc + e }
numbers.foldLeft(1, Int::times)
```

Idiomatic Kotlin code uses convention of having function parameters as last ones.

Generic functions

GenericFunctions.kt

```
fun <T : Comparable<T>>
    List<T>.maxOr(default: () -> T): T =
  maxOrNull() ?: default()
```

while compiler is smart enough to usually guess all needed types

Generic functions

GenericFunctions.kt

```
fun <T : Comparable<T>>
    List<T>.maxOr(default: () -> T): T =
  maxOrNull() ?: default()
```

while compiler is smart enough to usually guess all needed types

GenericFunctions.kt

```
listOf(3, 1, 4, 1, 5).maxOr { 92 }
```

Inline functions

InlineFunctions.kt

```
inline fun <reified T, V>
    Any?.letOrNull(f: (T) -> V) =
  if (this is T) f(this) else null
```

where the function parameter can have also `crossinline` or `noinline` modifiers

Kotlin

Inline functions

InlineFunctions.kt

```
inline fun <reified T, V>
    Any?.letOrNull(f: (T) -> V) =
  if (this is T) f(this) else null

"Kotlin".letOrNull<String, Int> { it.length }
```

where the function parameter can have also `crossinline` or `noinline` modifiers

Inline functions

InlineFunctions.kt

```
inline fun <reified T, V>
    Any?.letOrNull(f: (T) -> V) =
  if (this is T) f(this) else null

"Kotlin".letOrNull<String, Int> { it.length }

"Kotlin".letOrNull { str: String -> str.length }
```

where the function parameter can have also `crossinline` or `noinline` modifiers

Perform transformations on it

ItMappers.kt

```
inline fun <T> T.also(f: (T) -> Unit): T {
  f(this)
  return this
}

inline fun <T, R> T.let(f: (T) -> R): R {
  return f(this)
}
```

Perform transformations on it - examples

ItMappers.kt

```
val maxVal = listOf(3, 1, 4)
  .maxOrNull()
  .also(::println)

val doubled = maxVal?.let { it * 2 }
```

Perform transformations on this

ThisMappers.kt

```
inline fun <T> T.apply(f: T.() -> Unit): T {
  this.f()
  return this
}
```

Perform transformations on `this`

ThisMappers.kt

```
inline fun <T> T.apply(f: T.() -> Unit): T {
  this.f()
  return this
}

val server = JettyServer().apply {
  install(ForceHttps)
  install(Cors)
  start()
}
```

Tail recursion

TailRecursiveFunctions.kt

```
tailrec fun factorial(c: UInt, r: UInt): UInt =
  if (c > 1U) factorial(c - 1U, r * c) else r
```

Optimized by compiler to standard `while` loop pattern.

Tail recursion

TailRecursiveFunctions.kt

```
fun factorial(n: UInt): UInt {
  tailrec fun go(c: UInt, r: UInt): UInt =
    if (c > 1U) go(c - 1U, r * c) else r
  return go(n, 1U)
}
```

Optimized by compiler to standard `while` loop pattern.

Classes in Kotlin

In Kotlin we can define classes similarly as in Java but also we can define:

Classes in Kotlin

In Kotlin we can define classes similarly as in Java but also we can define:

- multiple classes in single file

Kotlin

Classes in Kotlin

In Kotlin we can define classes similarly as in Java but also we can define:

- multiple classes in single file
- extension functions in classes

Classes in Kotlin

In Kotlin we can define classes similarly as in Java but also we can define:

- multiple classes in single file
- extension functions in classes
- `sealed` classes as well as `enum` classes

Classes in Kotlin

In Kotlin we can define classes similarly as in Java but also we can define:

- multiple classes in single file
- extension functions in classes
- `sealed` classes as well as `enum` classes
- `objects` being singleton classes

Classes in Kotlin

In Kotlin we can define classes similarly as in Java but also we can define:

- multiple classes in single file
- extension functions in classes
- `sealed` classes as well as `enum` classes
- `objects` being singleton classes
- `typealiases` resolved at compile-time

Classes in Kotlin

In Kotlin we can define classes similarly as in Java but also we can define:

- multiple classes in single file
- extension functions in classes
- `sealed` classes as well as `enum` classes
- `objects` being singleton classes
- `typealiases` resolved at compile-time
- inline classes as `value classes`

Classes in Kotlin

In Kotlin we can define classes similarly as in Java but also we can define:

- multiple classes in single file
- extension functions in classes
- `sealed` classes as well as `enum` classes
- `objects` being singleton classes
- `typealiases` resolved at compile-time
- inline classes as `value classes`
- `inner classes` as well as nested `classes`

Base classes

BasicClasses.kt

```
abstract class Animal constructor(
  val color: Color, val owner: Person
) : Comparable<Animal> {
  enum class Color { RED, GREEN, BLUE }

  override fun compareTo(other: Animal) =
    color.compareTo(other.color)
}
```

Kotlin

Data classes

DataClasses.kt

```
data class Person(val name: String, val age: Int)

val alice = Person("Alice", 42)
val equal = alice == Person("Bob", 22)
val olderAlice = alice.copy(age = 66)
```

Data classes

DataClasses.kt

```
data class Person(val name: String, val age: Int)

val alice = Person("Alice", 42)
val equal = alice == Person("Bob", 22)
val olderAlice = alice.copy(age = 66)

val (name, age) = alice
println("Hello, I'm $name and I'm $age")
```

Data classes

DataClasses.kt

```
data class Person(val name: String, val age: Int)

val alice = Person("Alice", 42)
val equal = alice == Person("Bob", 22)
val olderAlice = alice.copy(age = 66)

with(alice) {
  println("Hello, I'm $name and I'm $age")
}
```

Example: Interpreter classes structure

InterpreterClasses.kt

```
typealias Ident = String
typealias Env = MutableMap<Ident, Int?>
typealias IntOp = (Int, Int) -> Int
interface Exp { fun eval(env: Env): Int? }

sealed class BinOp(val op: IntOp)
abstract class AddOp(op: IntOp) : BinOp(op)
abstract class MulOp(op: IntOp) : BinOp(op)
```

Kotlin

Example: Interpreter classes structure

InterpreterClasses.kt

```
object Plus : AddOp(Int::plus)
object Minus : AddOp(Int::minus)
object Times : MulOp(Int::times)
object Divide : MulOp(Int::div)
```

Example: Interpreter classes structure

InterpreterClasses.kt

```
data class BinExp(
  val l: Exp, val op: BinOp, val r: Exp
): Exp {
  override fun eval(env: Env): Int? {
    val lValue = l.eval(env) ?: return null
    val rValue = r.eval(env) ?: return null
    return op.op(lValue, rValue)
  }
}
```

Kotlin

Example: Interpreter classes structure

InterpreterClasses.kt

```
@JvmInline
value class NumExp(val value: Int): Exp {
  override fun eval(env: Env): Int = value
}

data class IdentExp(val i: Ident): Exp {
  override fun eval(env: Env): Int? = env[i]
}
```

Kotlin

Example: Interpreter classes structure

InterpreterClasses.kt

```
sealed class Stmt
class ExpStmt(val exp: Exp) : Stmt()
class AssStmt(val i: Ident, val e: Exp) :
  Stmt()
```

Example: Interpreter classes structure

InterpreterClasses.kt

```kotlin
fun step(env: Env, stmt: Stmt) = when(stmt) {
  is ExpStmt -> env.also {
    println(stmt.exp.eval(it))
  }
  is AssStmt -> env.apply {
    put(stmt.i, stmt.e.eval(env))
  }
}
```

Kotlin

Example: Interpreter classes structure

InterpreterClasses.kt

```
data class Program(
  private val stmts: MutableList<Stmt>
  = mutableListOf()
) {
  fun interpret(env: Env = mutableMapOf()) =
    stmts.fold(env, ::step)

  // we could stop there but...
```

Example: Interpreter classes structure

InterpreterClasses.kt

```
  // let's revisit functions knowledge
  infix fun Ident.eq(e: Exp) {
    stmts.add(AssStmt(this, e))
  }
  operator fun Exp.not() {
    stmts.add(ExpStmt(this))
  }
}
```

Example: Interpreter classes structure

InterpreterClasses.kt

```
operator fun Exp.plus(e: Exp) =
  BinExp(this, Plus, e)
operator fun Exp.times(e: Exp) =
  BinExp(this, Times, e)

inline fun def(i: Ident) = IdentExp(i)
inline fun num(i: Int) = NumExp(i)
```

Kotlin

Interpreter DSL usage example

InterpreterClasses.kt

```
val program = Program().apply {
  "x" eq num(42)
  "y" eq num(24)
  "z" eq def("x") + def("y")
  !def("z")
  !(def("z") * def("u"))
}
program.interpret()
println(program)
```

Kotlin

Interpreter DSL usage example

InterpreterClasses.kt

```
val program = Program().apply {
  "x" eq num(42)
  "y" eq num(24)
  "z" eq def("x") + def("y")
  !def("z")
  !(def("z") * def("u"))
}
program.interpret()
println(program)
```

It was less than 100 short lines of code

Kotlin

Interpreter DSL usage example

InterpreterClasses.kt

```
val program = Program().apply {
  "x" eq num(42)
  "y" eq num(24)
  "z" eq def("x") + def("y")
  !def("z")
  !(def("z") * def("u"))
}
program.interpret()
println(program)
```

It was less than 100 short lines of code but the effect is amazing.

Destructuring declarations

PairDestructuring.kt

```
val keyValue = "key" to Unit
val (k, v) = keyValue
```

Destructuring declarations

PairDestructuring.kt

```
val keyValue = "key" to Unit
val (k, v) = keyValue

// which corresponds to
val desugaredK = keyValue.component1()
val desugaredV = keyValue.component2()
```

where these fuctions can be defined as operators and are also predefined e.g. for lists and `data classes`.

Destructuring in lambdas

LambdaDestructuring.kt

```
{ a -> ... } // one parameter
{ a, b -> ... } // two parameters
{ (a, b) -> ... } // a destructured pair
{ (a, b), c -> ... } // pair and parameter
```

Delegated properties

Classes having defined `operator` functions `getValue` and `setValue`

Observables.kt

```
import kotlin.properties.Delegates.observable

class User {
  var name: String by observable("noname") {
    _, old, new -> println("$old -> $new")
  }
}
```

Delegated properties

The most commonly used is probably `lazy` initializer

LazyProperties.kt

```
class Book {
  val description: String by lazy {
    "value downloaded from database"
  }
}
```

which in other languages usually have to be defiend manually.

Ranges

There is only single range operator in language

Ranges.kt

```
val include = 1..42
val notInclude = 1 until 42
val down = 42 downTo 1
val collection = listOf(1, 2, 3)
for (i in collection.indices)
        println("At $i got ${collection[i]}")
```

Kotlin

Ranges

There is only single range operator in language

Ranges.kt

```
val include = 1..42
val notInclude = 1 until 42
val down = 42 downTo 1
val collection = listOf(1, 2, 3)
for (i in collection.indices)
        println("At $i got ${collection[i]}")

for ((i, v) in collection.withIndex())
        println("At $i got $v")
```

Kotlin

"Not implemented" marker

TODO.kt

```
fun `does P = NP`(): Boolean =
  TODO("Waiting for proof")
```

"Not implemented" marker

TODO.kt

```
fun `does P = NP`(): Boolean =
  TODO("Waiting for proof")
```

Not implemented function can have proper signature. It is part of standard library, not of the language.

"Not implemented" marker

TODO.kt

```
fun `does P = NP`(): Boolean =
  TODO("Waiting for proof")

inline fun TODO(reason: String): Nothing =
  throw NotImplementedError(
    "An operation is not implemented: $reason")
```

Not implemented function can have proper signature. It is part of standard library, not of the language.

Kotlin

Tricky lambda scope

Swap.kt

```
var a = 42
var b = 24
```

Tricky lambda scope

Swap.kt

```kotlin
var a = 42
var b = 24

val temp = a
a = b
b = temp
```

Tricky lambda scope

Swap.kt

```kotlin
var a = 42
var b = 24

a = b.also { b = a }
// now a = 24 and b = 42
```

Tricky lambda scope

Swap.kt

```
var a = 42
var b = 24

a = b.also { b = a }
// now a = 24 and b = 42
```

as in Kotlin variables captured in the closure can be modified in the lambda. It is not used commonly as `vars` aren't used.

Infrastructure

Kotlin Scripts

- Kotlin offers not only REPL for easy testing code but also provides mechanism of scripts

Kotlin Scripts

- Kotlin offers not only REPL for easy testing code but also provides mechanism of scripts
- they are standard source files with extension `.kts`

Kotlin Scripts

- Kotlin offers not only REPL for easy testing code but also provides mechanism of scripts
- they are standard source files with extension `.kts`
- can be run with `kotlinc` compiler but also custom solution like `kscript` exists

Kotlin Scripts

- Kotlin offers not only REPL for easy testing code but also provides mechanism of scripts
- they are standard source files with extension `.kts`
- can be run with `kotlinc` compiler but also custom solution like `kscript` exists
- defines the structure of `Gradle` project

Shell script

SimpleScript.kts

```
#!/usr/bin/env kscript
import java.io.File

val dir = args.singleOrNull() ?: "."
```

Shell script

SimpleScript.kts

```
#!/usr/bin/env kscript
import java.io.File

val dir = args.singleOrNull() ?: "."

File(dir)
  .walkTopDown()
  .filter(File::isFile)
  .sorted()
  .forEach(::println)
```

Kotlin DSL in Gradle

build.gradle.kts

```
plugins {
  kotlin("jvm") version "1.5.31"
}

// ...

tasks.withType<KotlinCompile> {
  kotlinOptions.jvmTarget = "1.8"
}
```

Kotlin Multiplatform

- can be compiled to native executables, JavaScript files and also to WebAssembly

Kotlin Multiplatform

- can be compiled to native executables, JavaScript files and also to WebAssembly
- features code sharing between all or some platforms used in project

Kotlin Multiplatform

- can be compiled to native executables, JavaScript files and also to WebAssembly
- features code sharing between all or some platforms used in project
- uses mechanism of `expect` and `actual` declarations

Kotlin Multiplatform

- can be compiled to native executables, JavaScript files and also to WebAssembly
- features code sharing between all or some platforms used in project
- uses mechanism of `expect` and `actual` declarations
- reuse the multiplatform logic in common and platform-specific code (it's even simpler when there are more and more multiplatform libraries)

expect and actual in action

Common.kts

```
// common
expect fun randomUUID(): String
```

expect and actual in action

Common.kts

```
// common
expect fun randomUUID(): String

// Android
actual fun randomUUID() =
   java.util.UUID.randomUUID().toString()
```

expect and actual in action

Common.kts

```
// common
expect fun randomUUID(): String

// Android
actual fun randomUUID() =
  java.util.UUID.randomUUID().toString()

// iOS
actual fun randomUUID(): String =
  platform.Foundation.NSUUID().UUIDString()
```

Multi languages interoperability

Kotlin Native uses LLVM backend for compiling to native binaries. It can create:

- executables for many platforms

Multi languages interoperability

Kotlin Native uses LLVM backend for compiling to native binaries.
It can create:

- executables for many platforms
- a static library or dynamic library with C headers for C/C++ projects

Multi languages interoperability

Kotlin Native uses LLVM backend for compiling to native binaries. It can create:

- executables for many platforms
- a static library or dynamic library with C headers for C/C++ projects
- an Apple framework for Swift and Objective-C projects

Multi languages interoperability

Kotlin Native uses LLVM backend for compiling to native binaries. It can create:

- executables for many platforms
- a static library or dynamic library with C headers for C/C++ projects
- an Apple framework for Swift and Objective-C projects

Kotlin supports using libraries from:

- static and dynamic C libraries

Multi languages interoperability

Kotlin Native uses LLVM backend for compiling to native binaries. It can create:

- executables for many platforms
- a static library or dynamic library with C headers for C/C++ projects
- an Apple framework for Swift and Objective-C projects

Kotlin supports using libraries from:

- static and dynamic C libraries
- Swift and Objective-C frameworks

Multi languages interoperability

Kotlin Native uses LLVM backend for compiling to native binaries. It can create:

- executables for many platforms
- a static library or dynamic library with C headers for C/C++ projects
- an Apple framework for Swift and Objective-C projects

Kotlin supports using libraries from:

- static and dynamic C libraries
- Swift and Objective-C frameworks

Let's try CLion or AppCode to test these great features if you're interested.

Kotlin Multiplatform project structure

Project defines hirarchical structure of sourcesets, for which the dependency relation is defined. E.g. we can have:

- `commonMain` source set containing all common logic

Kotlin Multiplatform project structure

Project defines hirarchical structure of sourcesets, for which the dependency relation is defined. E.g. we can have:

- `commonMain` source set containing all common logic
- `jvmMain:commonMain` code specific for JVM

Kotlin Multiplatform project structure

Project defines hirarchical structure of sourcesets, for which the dependency relation is defined. E.g. we can have:

- `commonMain` source set containing all common logic
- `jvmMain:commonMain` code specific for JVM
- `jsMain:commonMain` code specific for web

Kotlin Multiplatform project structure

Project defines hirarchical structure of sourcesets, for which the dependency relation is defined. E.g. we can have:

- `commonMain` source set containing all common logic
- `jvmMain:commonMain` code specific for JVM
- `jsMain:commonMain` code specific for web
- `desktopMain:commonMain` code specific for all desktop platforms

Kotlin Multiplatform project structure

Project defines hirarchical structure of sourcesets, for which the dependency relation is defined. E.g. we can have:

- `commonMain` source set containing all common logic
- `jvmMain:commonMain` code specific for JVM
- `jsMain:commonMain` code specific for web
- `desktopMain:commonMain` code specific for all desktop platforms
  - `linuxX64Main:desktopMain` code specific for Linux x64 platforms

Kotlin Multiplatform project structure

Project defines hirarchical structure of sourcesets, for which the dependency relation is defined. E.g. we can have:

- `commonMain` source set containing all common logic
- `jvmMain:commonMain` code specific for JVM
- `jsMain:commonMain` code specific for web
- `desktopMain:commonMain` code specific for all desktop platforms
  - `linuxX64Main:desktopMain` code specific for Linux x64 platforms
  - `macosX64Main:desktopMain` code specific for MacOS x64 platforms

Kotlin new compiler - K2

- Kotlin started with no native backend and no coroutines

Kotlin new compiler - K2

- Kotlin started with no native backend and no coroutines
- introduce new type inference algorithm

Kotlin new compiler - K2

- Kotlin started with no native backend and no coroutines
- introduce new type inference algorithm
- provide new JVM and JS IR backend

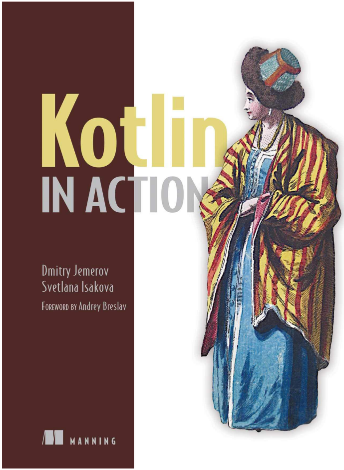
Kotlin new compiler - K2

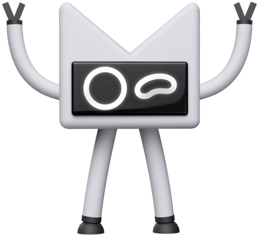
- Kotlin started with no native backend and no coroutines
- introduce new type inference algorithm
- provide new JVM and JS IR backend
- introducing FIR (frontend intermediate representation)

Kotlin new compiler - K2

- Kotlin started with no native backend and no coroutines
- introduce new type inference algorithm
- provide new JVM and JS IR backend
- introducing FIR (frontend intermediate representation)
- goal: performance improvements, provide an API for compiler extensions

Kotlin in Action - highly recommend





Thank you for your attention