Kotlin

101 + ½

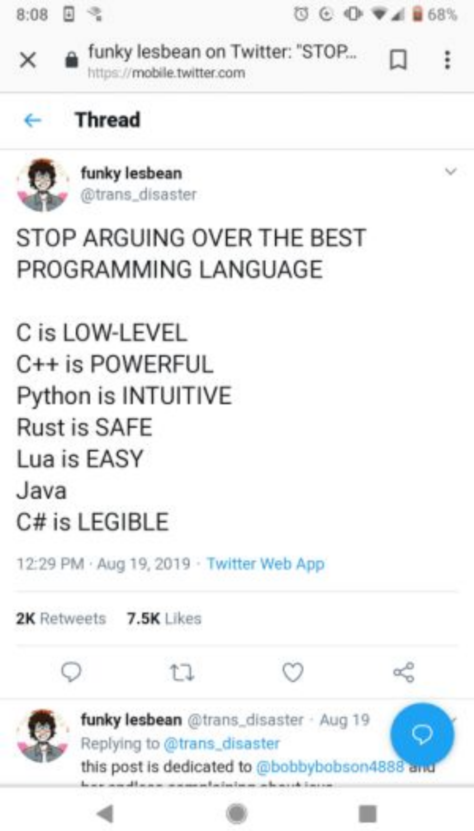
- Meet Kotlin
- 101 + ½
- Demo
- Coroutines

About Java

We don't want to be the first to include a feature, because every feature we add will never be removed.

Brian Goetz, Java Language Architect

About Java


8:08
68%
funky lesbean on Twitter: "STOP...
https://mobile.twitter.com
Thread

funky lesbean
@trans_disaster

STOP ARGUING OVER THE BEST PROGRAMMING LANGUAGE

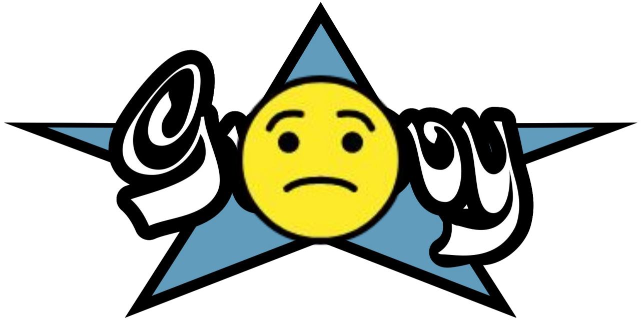
C is LOW-LEVEL
C++ is POWERFUL
Python is INTUITIVE
Rust is SAFE
Lua is EASY
Java
C# is LEGIBLE

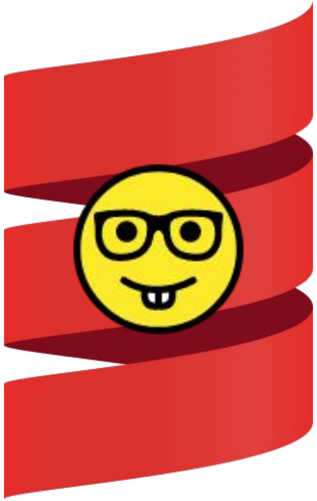
12:29 PM · Aug 19, 2019 · Twitter Web App

2K Retweets **7.5K** Likes

funky lesbean @trans_disaster · Aug 19
Replying to @trans_disaster
this post is dedicated to @bobbybobson4888 and

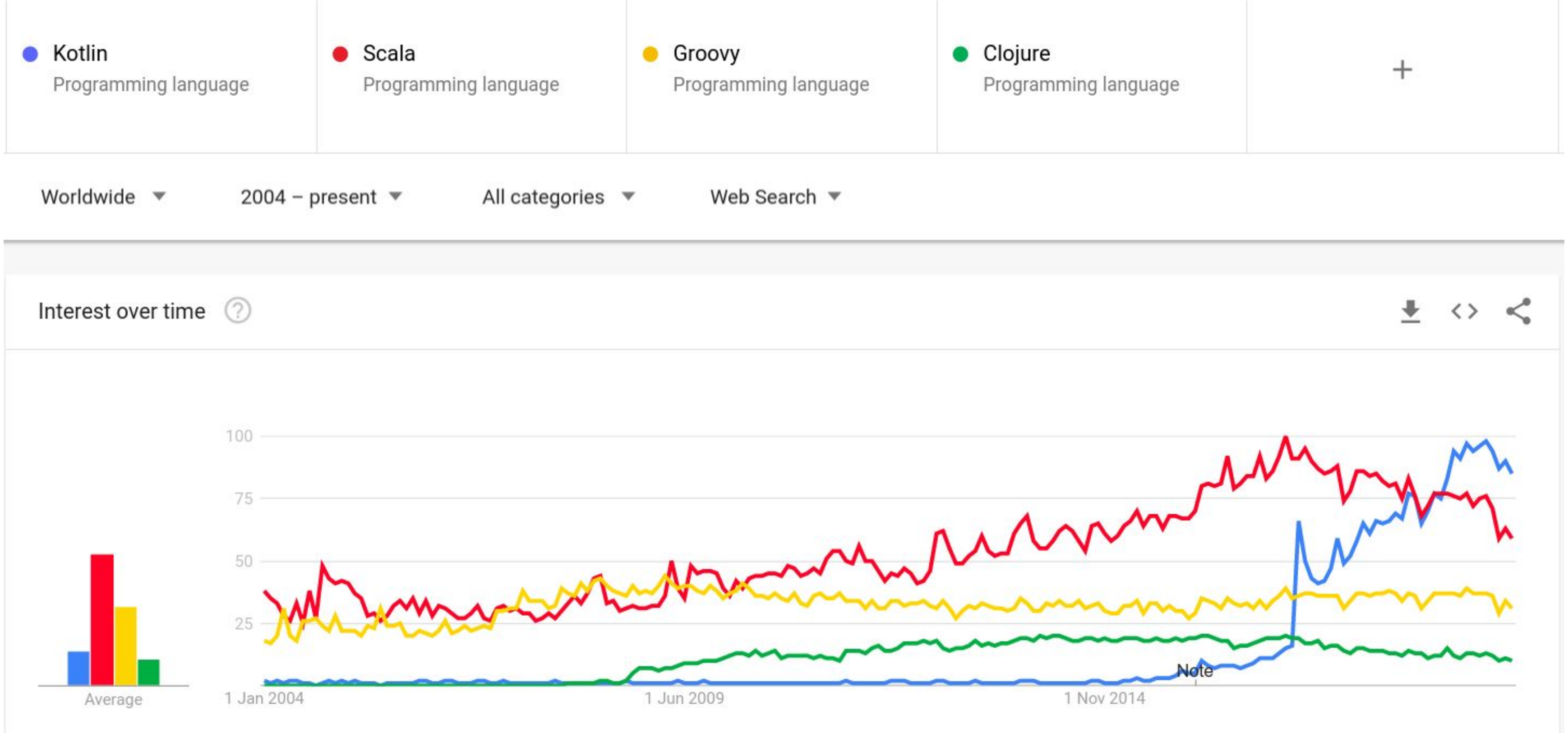
About Java





About Kotlin

About Kotlin



● Kotlin
Programming language

● Scala
Programming language

● Groovy
Programming language

● Clojure
Programming language

+

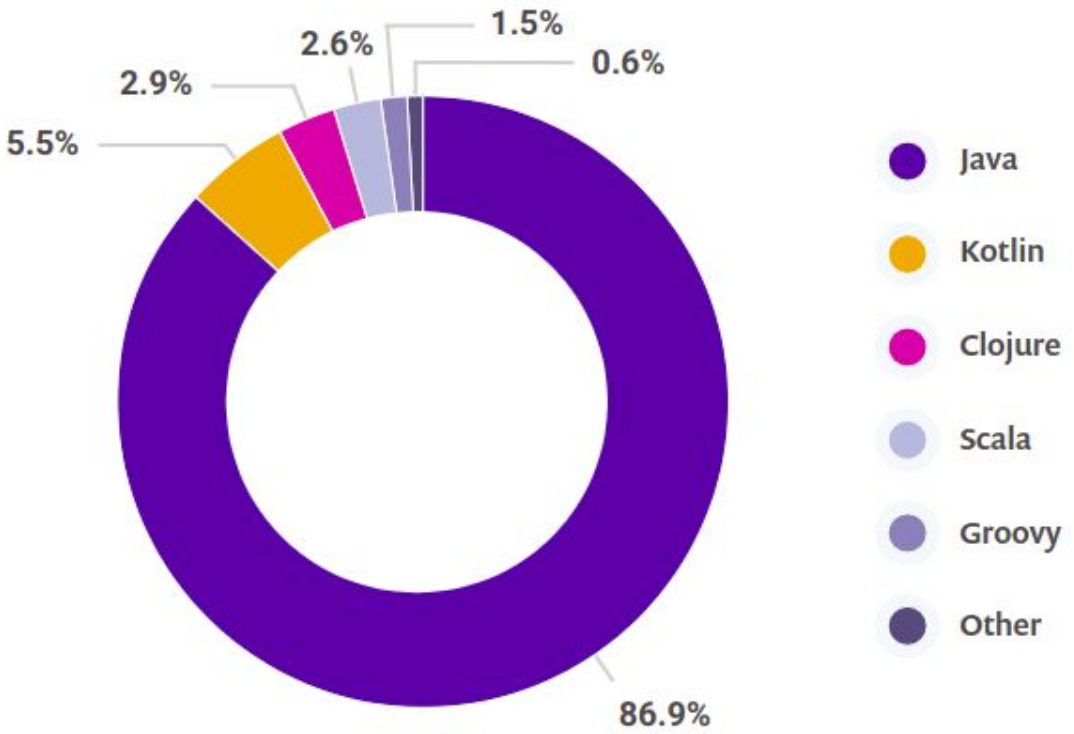
Worldwide ▾ 2004 – present ▾ All categories ▾ Web Search ▾

Interest over time

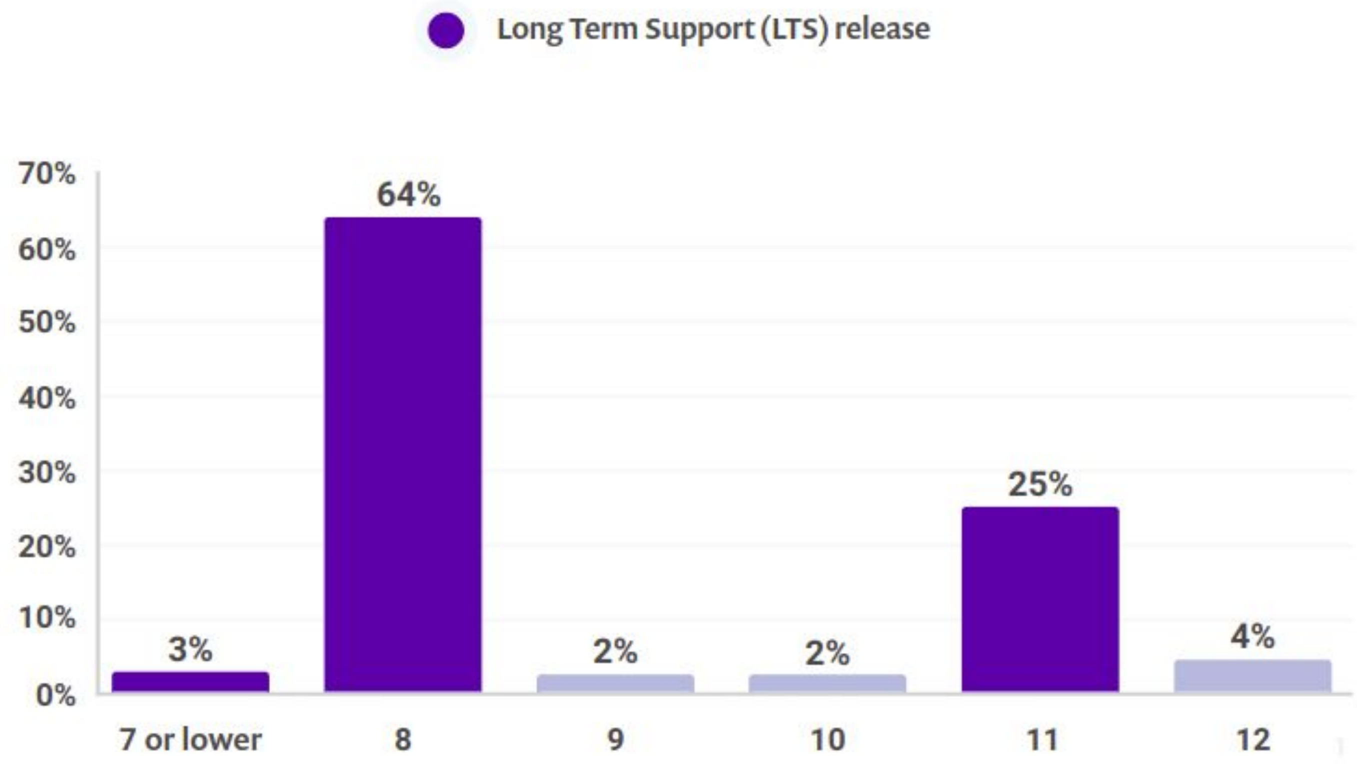
About Kotlin

About Kotlin

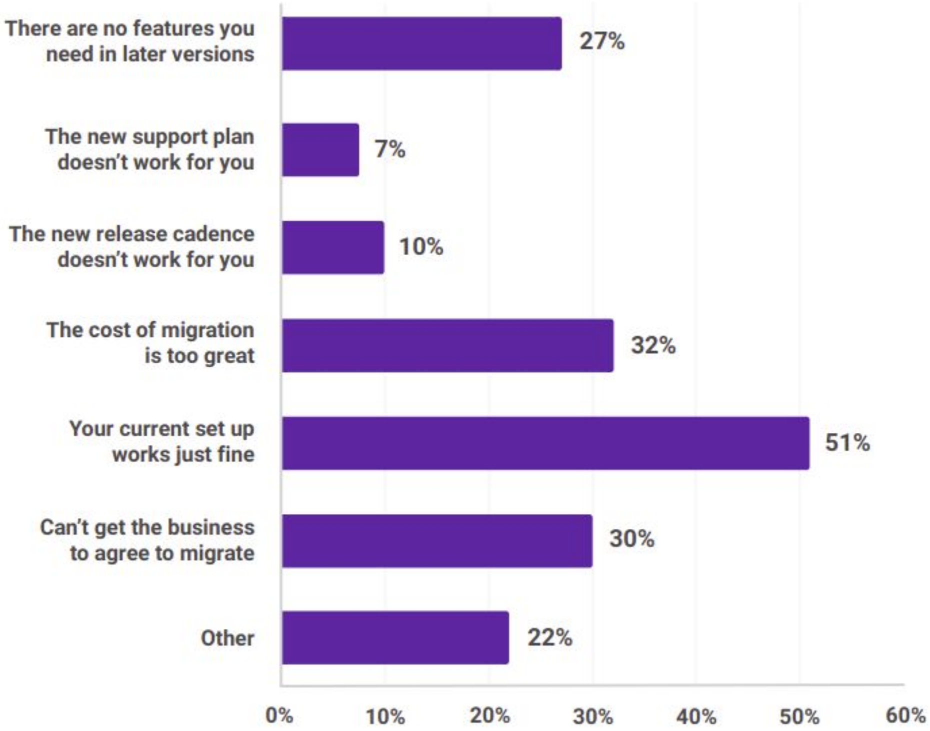
Snyk JVM Ecosystem Report 2020



Snyk JVM Ecosystem Report 2020



Snyk JVM Ecosystem Report 2020

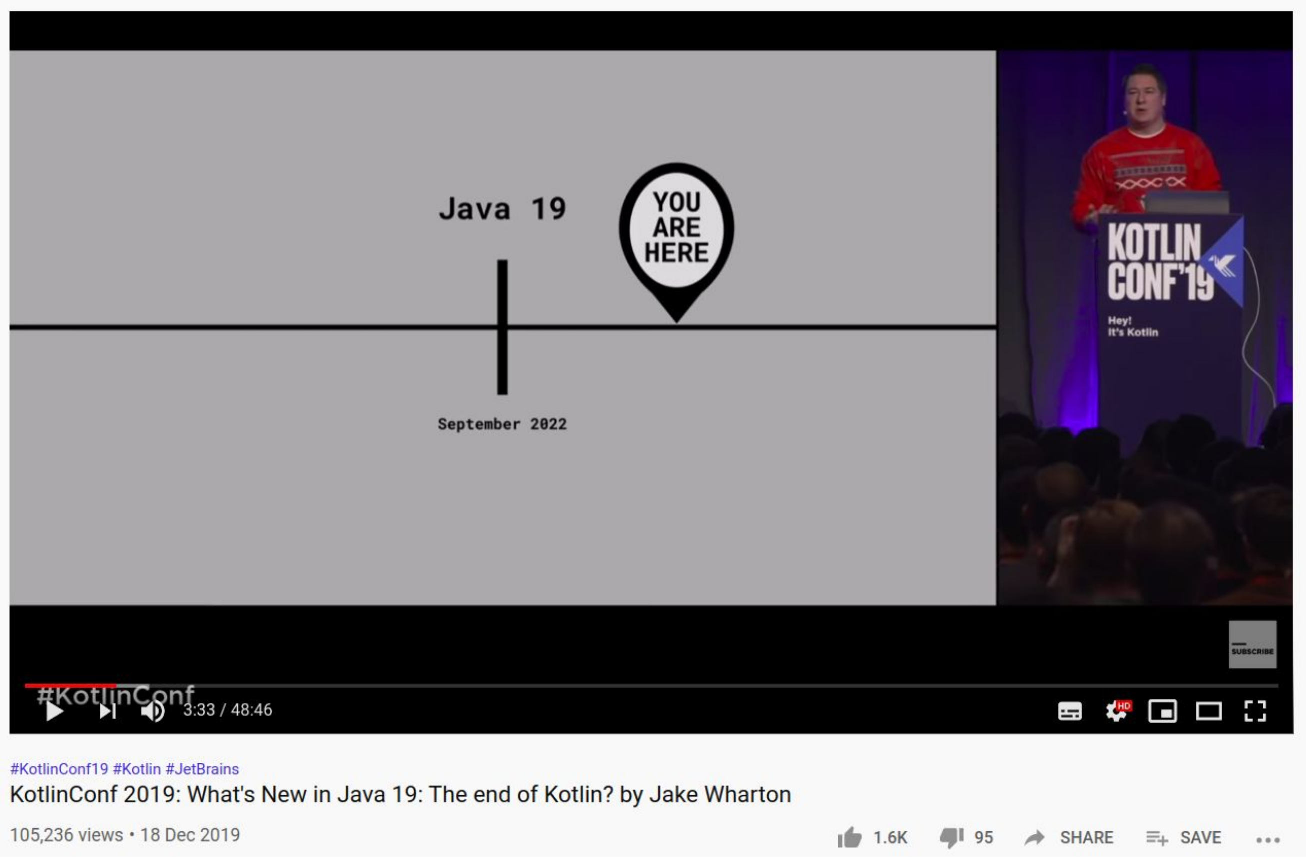


Up to 3 responses allowed.

Where Kotlin shines

- **Allows step by step migration**
- Increased productivity
- More readable code
- Safer code design and fewer error
- Easy to learn
- Great IDE support
- Great community and ecosystem (MockK, Arrow, Ktor, Kodein, TornadoFX, etc + Java ecosystem)
- Supported by Gradle
- Open-source with Apache 2 license
- Multiplatform* (JS, iOS, WebAssembly, Arm)

101



#KotlinConf19 #Kotlin #JetBrains

KotlinConf 2019: What's New in Java 19: The end of Kotlin? by Jake Wharton

105,236 views • 18 Dec 2019

101

;

101

;)

The `fun` keyword is used to declare a function. Programming in Kotlin is lots of fun, indeed!

Kotlin in Action, chapter 2.1.1

Kotlin 101 + ½

Destructuring declarations

creates multiple variables from a single data object

- arrays, pairs and triples

```
val (first, _, third) = arrayOf(1, 2, 3, 4, 5)
val (first, _, third) = Triple("first", 2, true)
```

- maps

```
for ((k, v) in map) { ... }
```

- data classes

```
data class Person(val name: String, val age: Int, val isMale: Boolean)
val (name, isMale) = Person("John", 18, true)
```

Type aliases

- provides alternative name for existing types

```kotlin
typealias FileTable<K> = MutableMap<K, MutableList<File>>
typealias MyHandler = (Int, String, Any) -> Unit
```

- compiler always expands the alias (not a new type)

```kotlin
typealias NamesByDepartments = Map<String, List<String>>
fun get ByDepartment(names: NamesByDepartments): List<String> {
   return names.getOrDefault("IT", listOf())
}
```

Ranges and Progressions

- provides arithmetic progressions for Int, Long and Char types

```
for (i in 1..4) print(i)
for (i in 'Z' downTo 'A' step 2) print(i)
for (i in 1 until 10) print(i)
```

- ranges are defined for comparable type and you can check if any instance is within the range

```
val versionRange = Version(1, 11)..Version(1, 30)
Version(0, 9) in versionRange
Version(1, 20) in versionRange
```

- it is possible to implement custom ranges by overloading ″`..`″ operator (`rangeTo`).

Collections and streams

- stream operations are instantly available on any collection

```
val result = listOf("one", "two", "three", "four")
  .mapIndexed { idx, value -> "${idx+1}${value.substring(1)}" } //[1ne,2wo,3hree,4our]
  .filter { it.length > 3 } //[3hree,4our]
  .reversed() //[4our,3hree]
```

- a lot of stream operations, i.e: `drop*, take*, reduce*, fold*, zip*, windowed, associate*, etc.`
- `sequenceOf(..)` provides the same operations but with lazy evaluation (huge or streaming collections)
- sequence generators can be used where generator stops when null is returned

```
val oddNumbersLessThan10 = generateSequence(1) { if (it < 10) it + 2 else null }
oddNumbersLessThan10.count() //6
```

or consumer requests no more items

```
val oddNumbers = sequence {
   yield(1)
   yieldAll(listOf(3, 5))
   yieldAll(generateSequence(7) { it + 2 })
}
oddNumbers.take(5).toList() //[1, 3, 5, 7, 9]
```

Collections and streams

- list specific operations, i.e.: `binarySearch`

```
class Product(val name:String, val price:Double)
val productList = listOf(
        Product("AppCode", 99.0),
        Product("WebStorm", 99.0),
        Product("DotTrace", 129.0))
productList.binarySearch(Product("WebStorm", 99.0),
                            compareBy<Product> { it.price }.thenBy { it.name }) //1
```

- set specific operations, i.e.:

```
val numbers = setOf(1, 2, 3)
numbers union setOf(4, 5) //[1,2,3,4,5]
numbers intersect setOf(2, 1) //[1,2]
numbers subtract setOf(3, 4) //[1,2]
```

- map specific operaionts, i.e.:

```
val numbersMap = mapOf("one" to 1)
numbersMap + mapOf("two" to 2, "one" to 11) //{one=11, two=2}
numbersMap - "one" //{}
```

Functions

- spread operator

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

- default arguments and named arguments

```
fun formatName(firstName: String, middleName: String ="", lastName: String): String {
... }
formatName(lastName = "Doe", firstName = "John")
```

- single-expression function

```
fun double(x: Int) = x * 2
```

- local functions

```
fun prefixDr(name: String):String {
    val prefix = "Dr."
    fun addPrefix(name: String) ="$prefix $name"
    return addPrefix(name)
}
```

Extensions

- extends any receiver type without inheriting

```
fun <T> List<T>.lastReversed(size: Int) =takeLast(size).reversed()
listOf(1,2,3,4).lastReversed(2) //[4, 3]
```

- extensions are resolved statically, without inserting new members into the receiver class (no overhead)

- member function with same signature always wins

- extension property (no backing field)

```
private val <T> List<T>.lastIndex: Int
   get() = size - 1

listOf(1,2).lastIndex //1
```

Classes

- primary constructor cannot contain any initialization logic (use init block)

```
open class Order(val itemName:String, val quantity:Int = 1, val price:Double) {
   val time = Instant.now()
   val total: Double
   init { total = quantity * price }
}
Order("Galaxy S10", price = 739.99) //goodbye builders
```

- all classes inherit from `Any` which implements `equals()`, `hashCode()` and `toString()`

- concise inheritance

```
class DozenOrder(name: String, price: Double): Order(name,12, price * 0.85)
```

- data classes also adds destructuring and copy method to clone object

```
data class Student(val name:String, val age:Int, val isMale:Boolean = true)
val student = Student("John", 17)
val (name, age) = student.copy(age = 18)
```

Sealed classes

- they are an extension of enum classes (can have multiple instance and inheritable)

```
sealed class Result(val result:Int)
data class Success(val data:Int): Result(data)
data class Error(val data:Int, val rootCause: Exception): Result(data)

fun notZeroResult(result: Result) =when(result) {
    is Success -> result.data
    is Error-> {
        val (data, rootCause) = result
        if (data != 0) data else { println(rootCause); null }
    }
}

val notZeros = listOf(
        Success(1),
        Error(0, IllegalArgumentException("conversion error"))
      )
        .mapNotNull(::notZeroResult) //[1]
```

Null safety

- compile time null safety provided by keywords, immutable types and helper methods
  `val, Any, Collection, listOf()`
  `var, Any?, MutableCollection, mutableListOf()`

- safe calls:
  `bob?.department?.head?.name`
  `bob?.let { println(it.department) }`

- Elvis operator: `val size = name?.length ?: -1`

- it is still possible to have `NullPointerException`, i.e.:
  - `throw NullPointerException()`
  - using non-null safe external Java code
  - using `!!` operator: `name!!.length`

Smart casts

- `is` checks and casts to desired type

```
if (any is String) print(any.length)
if (any is String && any.length > 0) print(any.length)
if (any !is String || any.length == 0) return
```

- `as` unsafe cast operator will throw `ClassCastException`if casting is not possible

```
val name: String = any as String
```

- `null` also can be casted

```
val name: String? = null as String?
```

- safe nullable cast will return `null` if cast is not possible

```
val name: String? = any as? String
```

Control flow expressions

- if is an expression returning values on the last expression of each block

```
val min = if (a < b) { println("a is smaller"); a }
   else { println("b is smaller"); b }
```

- try-catch is also an expression (finally will not have effect on the result)

```
val result: Double? = try { a/b }
   catch (ex: ArithmeticException) { null }
   finally { print("i'm done") }
```

- when is also an expression

```
val isBinary = when (sex.toLowerCase()) {
   "male" -> true
   "female" -> false
   else -> null
}
```

Lightweight pattern matching using `when`

```
when(any) {
   is Any -> TODO("any type")
   1 -> TODO("any value")
   2, 3 -> TODO("multiple value")
   in 'A'..'Z' -> TODO("within range")
   is EnumType -> TODO("is enum type")
   in EnumType.VALUE1..EnumType.VALUE3 -> TODO("within enum range")
   any == "someValue" || any == "otherValue" -> TODO("conditions")
   is SealedClass -> TODO("sealed classes (i.e. pattern matcher objects")
   in Regex("[0-9]") -> TODO("any type with overloaded operator 'contains'")
}

//Note: TODO is a built-in function returning Nothing
```

Nothing

- `Void` is a Java wrapper for `void`. It can be referenced from Kotlin but shouldn't.

```kotlin
fun foo(): Void? = null
```

- `Unit` is the Kotlin's equivalent of "no return anything" (`void`)

```kotlin
fun foo(): Unit = print("foo")
val foo = foo() //foo
print(foo) //kotlin.Unit
```

- `Nothing` means function doesn't return any value, not even `Unit`

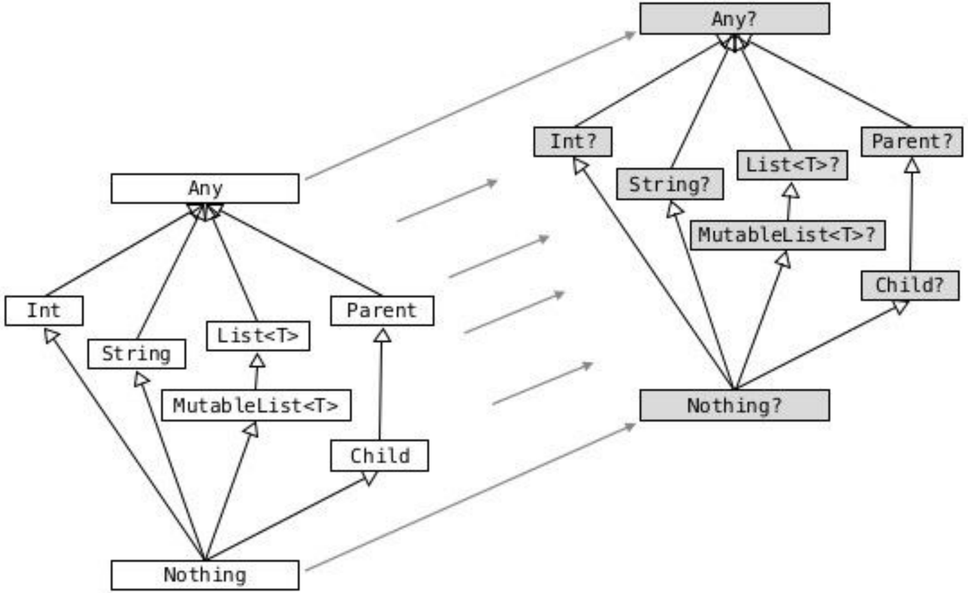
```kotlin
fun throwException(): Nothing = throw IllegalArgumentException()

fun foo(): String {
    throwException()
    return person.name // Compiler warns that this is unreachable code
}

fun bar(): String = person.name ?: throwException()
```

Nothing

Type hierarchy



Delegation

- “composition over inheritance” … but it hurts in Java

- Kotlin allows easy delegation using `by`

```
interface Shape { fun getShape(): String }
interface Color { fun getColor(): String }
class Triangle: Shape { override fun getShape() = "triangle" }
class Red: Color { override fun getColor() = "red" }

class ColoredShape(color: Color, shape: Shape): Colorby color, Shape by shape {
   fun show() { print("${this.getColor()} ${this.getShape()}") }
}

ColoredShape(Red(), Triangle()).show()
```

Delegation

- easy to use when need to use both Spring repository styles

```
interface SpringImplementedRepository : ReactiveMongoRepository<Any, String>

@Repository
class CustomMethodRepository(
        val repository: SpringImplementedRepository,
        val mongoTemplate: ReactiveMongoTemplate
): SpringImplementedRepository by repository {

   fun findByCustomLogic(value: Any): Flux<Any> =mongoTemplate…

}
```

Delegated properties

- operations on properties can be delegated as well

  - lazy computation and memoization

    ```
    val lazyValue by lazy { println("computed!");  "Hello" }
    ```

  - listening for a change of a property

    ```
    var name by observable("<no name>") { prop, old, new -> println("$old -> $new") }
    ```

  - storing properties in a map instead of separate fields

    ```
    class User(val map: Map<String, Any?>) {
       val name: String by map
       val age: Int     by map
    }
    val user = User(mapOf("name" to "John Doe", "age" to 25))
    ```

- create custom delegates by implementing operators below

  ```
  operator fun getValue(thisRef: Any?, property: KProperty<*>): String {..}
  operator fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {...}
  ```

Higher-Order Functions and Lambdas

- functions are first-class, so they can be stored in variables, passed as arguments and returned
- the format is: `(InputType1, InputType2) -> OutputType`
- parameter types are optional: `() -> OutputType`
- can have a receiver type: `ReceiverType.(InputType1) -> OutputType`
- names can be used for documenting meaning: `(x: Int, y:Int) -> Point`
- function types can be combined: `(Int) -> ((Int) -> Unit)`
- arrow notation is right associative: `((Int) ->(Int)) -> Unit`or `(Int) -> (Int) -> Unit`

```
val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
val sum: (Int, Int) -> Int = { x, y -> x + y }
val sum = {x,y -> x + y} //when types can be inferred
```

```
val repeatFun: String.(Int) -> String = { times -> this.repeat(times) }
//"hello".repeatFun(3) or repeatFun("hello", 3)
val twoParameters: (String, Int) -> String = repeatFun // OK

fun runTransformation(f: (String, Int) -> String) = f"(hello", 3)
val result = runTransformation(repeatFun) // OK
```

Domain-Specific Language

- member and extension functions can omit the dot and parentheses when marked with `infix`

```
infix fun Int.plus(second: Int) = this + second
```

`1.plus(2)` or `1 plus 2`

- the last function parameter can be placed outside the parentheses when it’s a function

```
fun runTransformation(f: (String, Int) -> String) = f("hello", 3)
val result = runTransformation { input, times -> input.repeat(times) }
```

- easy to build type-safe builders for markups, UI components, web server routes or test mocks
- helps to create readable tests by hiding boilerplate or complex logic (i.e. capturing)

```
captureSenderMessage { assertTrue(it.eventData.isNull) }

private fun captureSenderMessage(action: (OutboundMessage?) -> Unit) {
   argumentCaptor<Mono<OutboundMessage>>().apply{
       verify(senderMock, times(1)).send(capture())
       action(firstValue.block())
   }
}
```

Demo

Coroutines

- coroutines are light-weight threads which can suspend (and return) execution by calling other coroutine

```
runBlocking {
    launch {
        delay(1000L)
        println("World!")
    }
    println("Hello,")
}
```

```
fun main() = runBlocking {
    launch { doWorld() }
    println("Hello,")
}

suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
```

- launched jobs can be cancelled or can have timeout
- it is possible to cancel in the middle of computation if the coroutine checks for cancellation (`yield` or `isActive`)

```
val result = withTimeoutOrNull(1300L) {
    repeat(1000) { i -> println("I'm sleeping $i ..."); delay(500L) }
   "Done" // will get cancelled before it produces this result
}
```

Coroutines

- suspendable functions are sequential when not called as coroutine

```kotlin
suspend fun doSomethingUsefulOne(): Int { delay(1000L); return 13 }
suspend fun doSomethingUsefulTwo(): Int { delay(1000L); return 29 }
```

```kotlin
val time = measureTimeMillis {
    val one = doSomethingUsefulOne()
    val two = doSomethingUsefulTwo()
    println("${one + two}")
} // 2 sec
```

```kotlin
val time = measureTimeMillis {
    val one = async { doSomethingUsefulOne() }
    val two = async { doSomethingUsefulTwo() }
    println("${one.await() + two.await()}")
} // 1 sec
```

- `async` returns a `Deferred` (a promise) and `.await()` get it’s eventual result

- cancellation is always propagated through coroutines hierarchy

Channels

- `Deferred` transfers single value, Channels transfer a stream of values
- `Channel` is similar to a `BlockingQueue`, but it works with a suspending `send` and `receive` methods

```
val channel = Channel<Int>()
launch {
    for (x in 1..5) channel.send(x * x)
    channel.close()
}
for (y in channel) println(y)
println("Done!")
```

- this can be replaced with `produce` builder and `consumeEach` extension function

```
fun CoroutineScope.produceSquares(): ReceiveChannel<Int> =produce {
    for (x in 1..5) send(x * x)
}
fun main() = runBlocking {
    val squares = produceSquares()
    squares.consumeEach { println(it) }
    println("Done!")
}
```

Channels

- pipelines can be constructed using channels (also fan-out and fan-in)

```
suspend fun sendString(channel: SendChannel<String>, s: String, time: Long) {
    while (true) { delay(time); channel.send(s) }
}

val channel = Channel<String>()
launch { sendString(channel, "foo", 200L) }
launch { sendString(channel, "BAR!", 500L) }
repeat(6) { println(channel.receive()) }
coroutineContext.cancelChildren() // to cancel new coroutines
```

- buffered and ticker channels are also available

- select expression is experimental

Flow

- it is possible to return multiple asynchronous values using Flows

```
fun foo(): Flow<Int> = flow { //flow block can suspend
    for (i in 1..3) { delay(100); emit(i) }
}

fun main() = runBlocking<Unit> {
    foo().collect { value -> println(value) }
}
```

- flows are cold streams, started only when a terminal operation called (every time, i.e. `collect`)
- transformation operators (ie. `map`, `filter`, `zip`) can be used with suspending functions
- timeout and cancellation also works

```
runBlocking<Unit> {
    (1..3).asFlow().take(2).collect { value -> println(value) } //1 2
}
```

- some operators are still experimental
- Reactive Streams TCK compliant with Reactor and RxJava2 integration

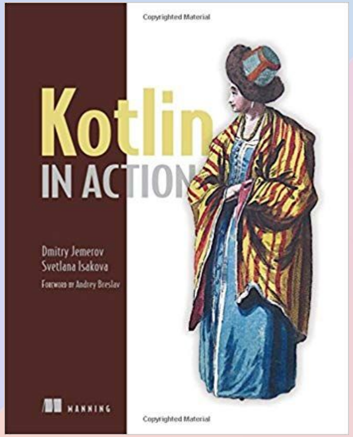
Demo

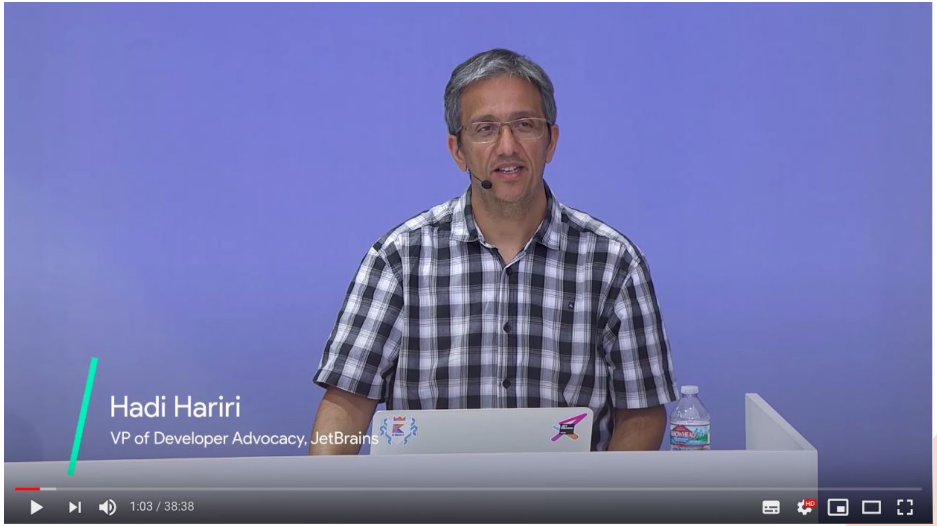
Resources

Demo: https://github.com/domahidizoltan/presentation-kotlin-101-0.5

Kotlin language guide: https://kotlinlang.org/docs/reference/

Dmitry Jemerov and Svetlana Isakova: Kotlin in Action





Jake Wharton: What's New in Java 19: The end of Kotlin?

any Hadi Hariri presentation:

- Introduction to Kotlin
- Kotlin 102 - Beyond the basics
- Functional Programming with Kotlin

Blog: Convincing Your Management to Introduce Kotlin

Thank you

?:

Zoltán Domahidi
Senior Kotlin/Java developer
domahidi.zoltan@telekom.hu