Kotlin 1.1

Christoph Pickl

Vienna, Austria – April 18th, 2017

Kotlin 1.1

language for JVM, Android & JS

(language for the masses)

Kotlin 1.1 is out

- Released on 1.3.2017 (about a year after 1.0)
- Current 1.1.1 bugfix release two weeks later
- Fully backwards compatible with 1.0
    - Commitment to **binary compatibility**
    - Kotlin `2.0`? – A possibility, not a plan
    - Migration tools will be provided
- Currently working on kotlin-native
- Gradle and Spring are using Kotlin

JVM News

- **Java 8** bytecode via compiler option: `-jvm-target 1.8`
- **JDK 8** classes via dependency: `kotlin-stdlib-jre8`
- Preserve **parameter names** via `-java-parameters`
- `const val` are now being **inlined** by the compiler
- Structural changes for `kotlin-reflect.jar` (Java 9)
  - `kotlin.reflect` renamed to `kotlin.reflect.full`
  - Old **deprecated** and to be removed with Kotlin `1.2`

Scripting Engine

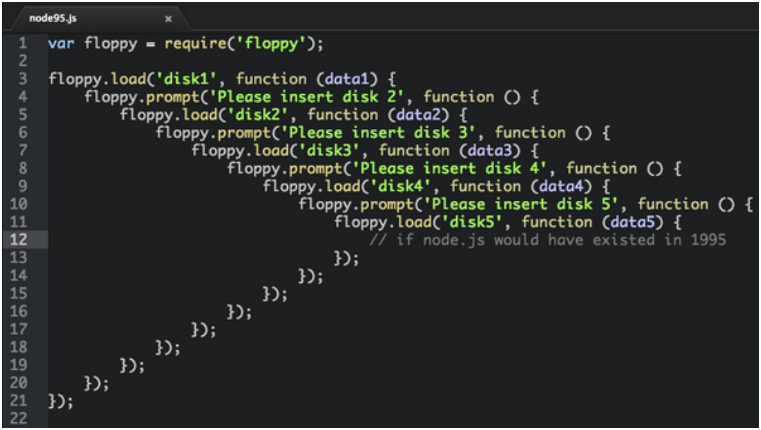
Requires some kts engine like: `kotlin-jsr223-local-example`

```kotlin
val engine = ScriptEngineManager()
  .getEngineByExtension("kts")
  ?: throw Exception("kts not supported!")

engine.eval("val x = 3")
println(engine.eval("x + 2"))  // 5
```

Language Features

What we'll cover

1. **Coroutines** (experimental)
2. Type **aliases**
3. Bound callable **references**
4. Improved **data** and **sealed** classes
5. **Destructuring** in lambdas
6. Local **delegated** properties
7. **Underscore** for numeral literals / unused parameters
8. Type inference and inlining for **properties**
9. Generic **enum** value access
10. Restrict lambda scope with `@DslMarker`



```
var floppy = require('floppy');

floppy.load('disk1', function (data1) {
    floppy.prompt('Please insert disk 2', function () {
        floppy.load('disk2', function (data2) {
            floppy.prompt('Please insert disk 3', function () {
                floppy.load('disk3', function (data3) {
                    floppy.prompt('Please insert disk 4', function () {
                        floppy.load('disk4', function (data4) {
                            floppy.prompt('Please insert disk 5', function () {
                                floppy.load('disk5', function (data5) {
                                    // if node.js would have existed in 1995
                                });
                            });
                        });
                    });
                });
            });
        });
    });
});

```

Source: https://collinmakersquare.wordpress.com

Coroutines are . . .

- very lightweight threads (like fibers)
- still marked as experimental
- very low-level designed so that frameworks can build upon it
  - Extensions for Android, JavaFX, . . .
  - like `async/await` from C# or `yield` from Python
- a very expressive tool for implementing asynchronous behavior
  - better syntax, look like regular function invocations

Coroutines async load

Coroutines scale pretty good processing loads of async operations.

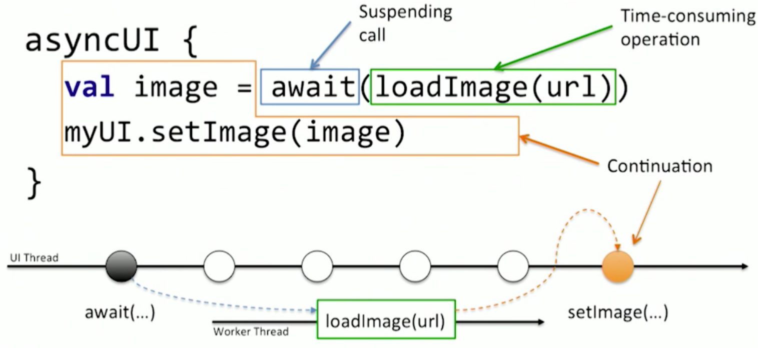
```kotlin
val jobs = List(100_000) {
  async(CommonPool) {
    delay(1000L)
    1
  }
}

runBlocking { // bridge async world
  println(jobs.sumBy {
    it.await()
  })
}
```

Threads instead

The same code with good old threads will . . .

```
for (i in 1..100_000) {
  thread(start = true) {
    Thread.sleep(1000)
  }
}
```

OutOfMemoryError: unable to create new native thread!



```
asyncUI {
   val image = await(loadImage(url))
   myUI.setImage(image)
}
```

Suspendable computation explained

Just some coroutine sample

```kotlin
suspend fun greet(name: String): String {
  delay(randomTime)
  return "Hello $name!"
}

runBlocking {
  listOf("foobar", "World").map { name ->
    async(CommonPool) {
      greet(name)
    }
  }.map { it.await() }.joinToString()
}
```

Type aliases

Add improved readability through custom naming.

```kotlin
typealias StatusCode = Int
fun request(expected: StatusCode) {}
request(200)

typealias Listener = (Event) -> Unit
fun subscribe(listener : Listener) {}
fun subscribe(listener : (Event) -> Unit) {}

class Outer {
  typealias Nope = Double // compile error
}
```

Bound callable references

Reference a member of an object instance.

```kotlin
val numberRegex = "\\d+".toRegex()
val list = listOf("a", "1")

// before kotlin11
list.filter { numberRegex.matches(it) }
  .forEach(::println)

// with kotlin11
list.filter(numberRegex::matches)
  .forEach(::println)
```

(KEEP)

Improved data and sealed classes

Subtypes outside of `sealed` class and `data` class inheritance added.

```kotlin
sealed class Expression

// sealed subs can be declared outside
object Operator : Expression()

// a data class can now extend another class
data class Operand(val symbol: String)
  : Expression()
```

Destructuring in lambdas

Destructuring now works in lambdas (and for data class).

```kotlin
val map = mapOf(1 to "one")

// before kotlin11
map.mapValues { entry ->
  val (key, value) = entry
  "$key = $value"
}

// with kotlin11
map.mapValues { (key, value) ->
  "$key = $value" }
```

Local delegated properties

Not only for (class) properties, anymorebut also for local variables.

```kotlin
fun exec(stringProvider: () -> String) {
  val string by lazy(stringProvider)

  // short circuit evaluation FTW
  if (condition() && string.isValid()) {
    // computed result will be cached
    println(string)
  }
}
```

(KEEP)

Underscores

Underscore for numeral literals and unused parameters.

```
val creditCardNumber = 1234_5678_9012_3456L
val hexBytes = 0xFF_EC_DE_5E
val hexWords = 0xCAFE_BABE
val maxval = 0x7fff_ffff_ffff_ffffL
val bytes = 0b0110_1001;

mapOf(1 to "one").map { (k, _) -> k }
```

(KEEP)

Enhanced Properties

Type inference and inlining for **properties**

```
data class Person(val age: Int) {
  // before kotlin11
  val isAdult get(): Boolean = age >= 18

  // with kotlin11
  val isAdult get() = age >= 18
}

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

(KEEP)

Generic enum value access

Supported via `enumValueOf()` and `enumValues()`.

```kotlin
enum class RGB { RED, GREEN, BLUE }

val red: RGB = enumValueOf("RED")
val rgbs: Array<RGB> = enumValues()

println(enumValues<RGB>()
  .joinToString(transform = RGB::name))
  // RED, GREEN, BLUE
```

(KEEP)

DSLs – The Problem

The inner scope always inherits the context of the outer scope:

```
html {
  head {
    head {
      // this should not be able
      // as it makes no sense
    }
  }
}
```

When within `head` let's remove the context of `html`!

Casual implementation of HTML DSL

```kotlin
fun html(code: HtmlContext.() -> Unit) {
  code(HtmlContext())
}
class HtmlContext {
  fun head(code: HeadContext.() -> Unit) {
    code(HeadContext())
  }
}
class HeadContext { }
```

Restrict scope with @DslMarker

```kotlin
fun html(code: HtmlContext.() -> Unit) {
  code(HtmlContext())
}
@MyDslMarker class HtmlContext {
  fun head(code: HeadContext.() -> Unit) {
    code(HeadContext())
  }
}
@MyDslMarker class HeadContext { }
@DslMarker annotation class MyDslMarker
```

(KEEP)

Humans still remain in control

If you still want to access the `html` context you can still do so.

```
html {
  head {
    // head { } compile error!
    this@html.head { } // enforce
  }
}
```

Standard Library

What we'll cover

1. Nullable number conversion
2. `also()`, `takeIf()`, `takeUnless()`
3. `minOf()`, `maxOf()`
4. `onEach()`
5. `groupingBy()`
6. Map functions
7. List comprehension
8. Array manipulation
9. Base classes for collections
10. `mod` renamed to `rem`

Nullable number conversion

Conversion avilable for: Byte, Short, Int, Long, Float, Double

```
"x".toIntOrNull() ?: 42 // = 42

// delegates to java.lang.Integer.parseInt
"x".toInt() // NumberFormatException

val radix = 2 // 2..36
// radix2 = 101010, radix16 = 2a
println(42.toString(radix))
// radix2 = 3, radix16 = 17
println("11".toIntOrNull(radix))
```

(KEEP)

also()

Same as apply() but without changing the this reference.

```kotlin
val a1 = "a".apply {
  "b".apply {
    this // "b"
    @Suppress("LABEL_NAME_CLASH")
    this@apply // "b" but want "a" :(
  }
}
val a2 = "a".also { outer ->
  "b".also {
    it // "b"
    outer // "a" :)
  }
}
```

takeIf() and takeUnless()

takeIf() is like filter() but acts on a singlue value and returns null on mismatch. takeUnless() simply inverts the condition.

```
val file = File("path")
if (!file.exists()) {
  return false
}

// takeIf works well with elvis operator
val file = File("path").takeIf(File::exists)
  ?: return false
```

Keyword highlighter using `takeIf()`

```kotlin
val input = "Kotlin"
val keyword = "in"

val index = input.indexOf(keyword)
    .takeIf { it >= 0 } ?: error()
//  .takeUnless { it < 0 } ?: error()

println("'$keyword' was found in '$input'")
println(input)
println(" ".repeat(index) + "^")
// 'in' was found in 'Kotlin'
// Kotlin
//     ^
```

minOf() and maxOf()

```kotlin
val l1 = listOf("one")
val l2 = listOf("0", "1")
val l3 = listOf("x", "y", "z")

// delegates to Math.min for 2 values
minOf(l1.size, l2.size)
Math.min(l1.size, l2.size)

minOf(l1.size, l2.size, l3.size)
Math.min(l1.size, Math.min(l2.size, l3.size))

// minOf/maxOf supports only 2-3 values :-/

minOf(l1, l2, compareBy { it.size })
```

So why not simply use: `Iterable<T>.min(): T?`

onEach()

```
fun <T, I : Iterable<T>> I.onEach(action:  (T) -> Unit):  I
fun <T> Iterable<T>.forEach(action:  (T) -> Unit):  Unit
```

```
listOf("foobar", "foo")
  .filter { it.endsWith("bar") }
  // chain item processing
  .onEach { println("Found item: $it") }
  .forEach { /* finally operate on them */ }
```

(KEEP, same as `apply { forEach { } }`)

groupingBy()

```
fun <T, K> Iterable<T>.groupingBy(key:  (T) -> K): Grouping<T, K>
fun <T, K> Iterable<T>.groupBy(key:  (T) -> K): Map<K, List<T>>
```

```
val list = listOf("anna", "otto", "oscar")

list
  .groupBy(String::first)
  .mapValues { (_, list) -> list.size } // a=1, o=2
  // creates intermediate map

list
  .groupingBy(String::first)
  .eachCount() // invokes foldTo()
```

Map functions

toMap(), toMutableMap(), minus, getValue(), withDefault()

```kotlin
var map = mapOf("x" to 1)
map.toMap() // create a copy
map.toMutableMap() // create a mutable copy

// plus already worked
map += ("y" to 2)
// now minus also supported
map -= "y"

map.getValue("y") // throws
val map2 = map.withDefault { "!\$it!" }
map2.getValue("y") // !y!
```

(KEEP)

Something I'm missing here . . .

Create a mutable (!) map based on a list of pairs.

```
val list: Pair<String, Int> = listOf("x" to 1)
list.toMap() // already existing
list.toMutableMap() // does NOT exist

// let's workaround
fun <K, V> Iterable<Pair<K, V>>.toMutableMap(): M
  val immutableMap = toMap()
  val map = HashMap<K, V>(immutableMap.size)
  map.putAll(immutableMap)
  return map
}
```

List comprehension

```
fun <T> List(size: Int, init: (index: Int) -> T): List<T>
```

```
// already existed for arrays
IntArray(4) { it * 2 }.toList()
// [0, 2, 4, 6]

// now for lists as well
List(4) { it * 2 }
MutableList(4) { it * 2 }
```

(Still not the same as in Haskell)

Array manipulation

New methods: `content[Deep](Equals|HashCode|ToString)`

```
val a1 = arrayOf("a", "b")
val a2 = arrayOf(arrayOf("a"), arrayOf("b"))

a1.toString() // [Ljava.lang.String;\@1b3af
a1.contentToString() // [a, b]
a2.contentDeepToString() // [[a], [b]]

// ... equals, hashCode the same ...
```

(Actually just a delegation to `java.util.Arrays`)

Base classes for collections

New classes: `Abstract[Mutable](Collection|List|Set|Map)`

```kotlin
// skeletal implementation of [List]
val listWithOneElement: List<String> =
  object : AbstractList<String>() {
    override val size: Int
      get() = 1
    override fun get(index: Int): String {
      return "always foo"
    }
  }
```

(See: KEEP, stdlib sources)

mod renamed to rem

mod function on integral types is inconsistent with BigInteger:

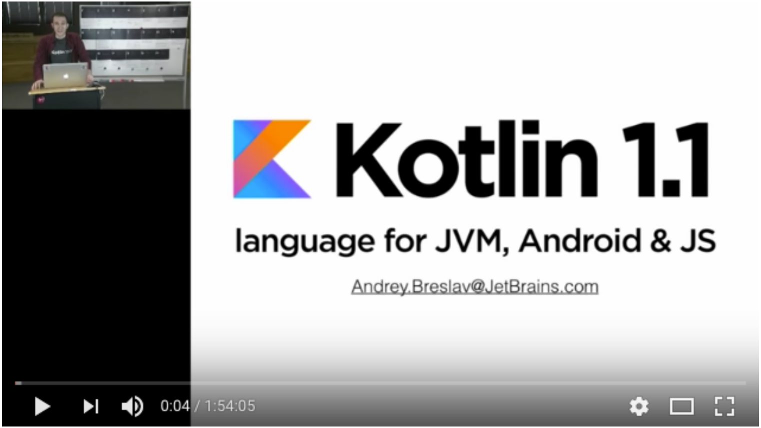
```
val minus3 = BigInteger.valueOf(-3)
val plus5 = BigInteger.valueOf(5)

minus3.mod(plus5) // 2
% Int.mod() was deprecated
minus3.toInt().mod(plus5.toInt()) // -3

minus3.rem(plus5) // -3
minus3.toInt().rem(plus5.toInt()) // -3
```

(Math nerds know their ecuclidean rings)

That’s it

Further Reading

- Slides and sources
  https://github.com/.../kotlin11slides
- Some sample code
  https://github.com/.../awesomekotlin/kotlin11
- Official release page
  `https://kotlinlang.org/docs/reference/whatsnew11.html`
- **K**otlin **E**volution and **E**nhancement **P**rocess
  `https://github.com/Kotlin/KEEP`
- Kotlin Vienna Usergroup
  `https://www.meetup.com/Kotlin-Vienna/`

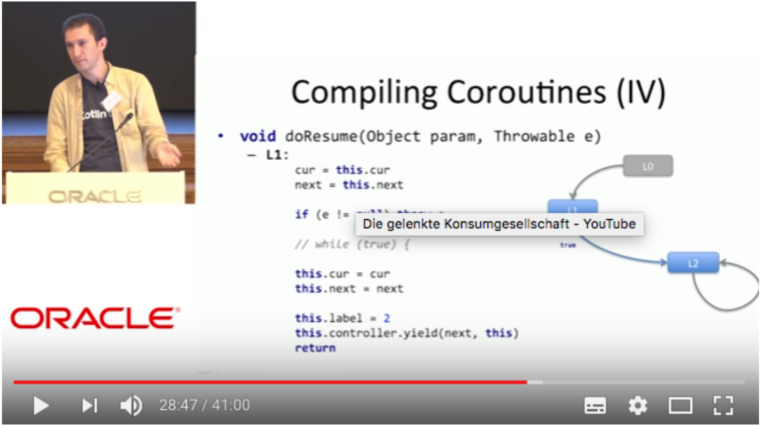


Kotlin 1.1

language for JVM, Android & JS

Andrey.Breslav@JetBrains.com

https://youtube.com/watch?v=zpyJHSR-5ts



Compiling Coroutines (IV)

- `void doResume(Object param, Throwable e)`
  - L1:

```
cur = this.cur
next = this.next

if (e != 

// while (true) {

this.cur = cur
this.next = next

this.label = 2
this.controller.yield(next, this)
return
```

Die gelenkte Konsumgesellschaft - YouTube

https://www.youtube.com/watch?v=4W3ruTWUhpw

One more thing . . .

Logging, the Kotlin way

First declare a Gradle dependency (Slf4j extensions):

```
compile
 "io.github.microutils:kotlin-logging:1.4.4"
```

Write your own shortcut function:

```
fun LOG(func: () -> Unit) =
  KotlinLogging.logger(func)
// define a code template in your IDE
```

Simple usage:

```
class Foo {
  private val log = LOG {}
  init {
    log.debug { "lazy evaluated $this" }
  }
}
```

Have a nice Kotlin :}