Get Started With Kotlin

DAY 1

- Brief Motivation
- Diving Into Kotlin
- Functional Programming

DAY 2

- Object-Orientation
- Where to Go From Here

Get Started With Kotlin

DAY 1

- **Brief Motivation**
- Diving Into Kotlin
- Functional Programming

DAY 2

- Object-Orientation
- Where to Go From Here

Brief Motivation

1. **Why Me?**
2. Why Kotlin?
3. Who‘s Using Kotlin?
4. What Can I Use It For?

About Me

- Peter Sommerhoff
- RWTH Aachen University, Germany
- Kotlin since 2015/2016
- Online Instructor (35,000+ students)
  - Passion for teaching
  - Teaching Kotlin since 2016
- Author of „Kotlin for Android App Development“
  - To be released end of 2018 (or early 2019)

Brief Motivation

- ✓ Why Me?
- **2. Why Kotlin?**
- 3. Who‘s Using Kotlin?
- 4. What Can I Use It For?

Google I/O 17

Source: https://www.youtube.com/watch?v=d8ALcQiuPWs

StackOverflow Survey 2018

Most Loved, Dreaded, and Wanted

Source: https://insights.stackoverflow.com/survey/2018

Most Loved, Dreaded, and Wanted Languages

Loved | Dreaded | Wanted

- Survey of over 100,000 developers

HackerRank Survey 2018

Source: https://research.hackerrank.com/developer-skills/2018/

- Survey of almost 40,000 developers

RebelLabs Survey 2017

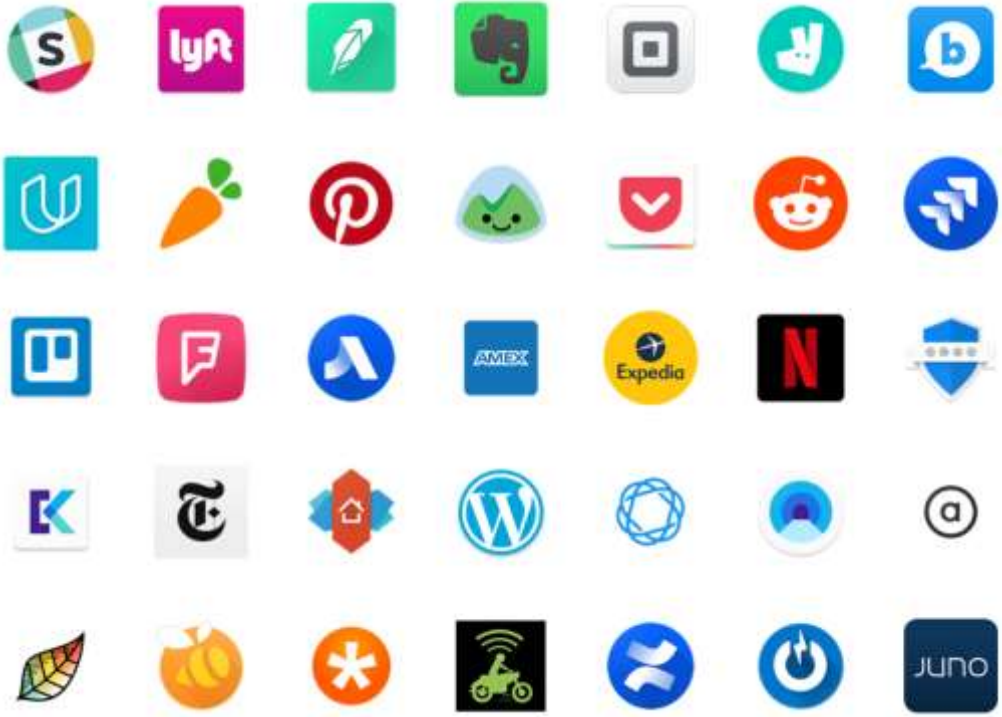
Source: https://zeroturnaround.com/rebellabs/developer-productivity-report-2017-why-do-you-use-java-tools-you-use/

- Survey of over 2000 developers

Brief Motivation

- ✓ Why Me?
- ✓ Why Kotlin?
- **3. Who‘s Using Kotlin?**
- 4. What Can I Use It For?

Kotlin on Android



Source: https://developer.android.com/kotlin/

Companies Using Kotlin

- …and of course many more
  - Google
  - Jetbrains
  - Amazon
  - Coursera
  - Foursquare
- Takeaways
  - Ready for production
  - Good experiences in practice

Brief Motivation

- ✓ Why Me?
- ✓ Why Kotlin?
- ✓ Who‘s Using Kotlin?
- **4. What Can I Use It For?**

Kotlin is Everywhere

- Kotlin = JVM language + more
  - Wherever Java is used + more!
- Android
- Desktop (e.g. JavaFX / Swing)
- Backend (e.g. Spring)
- Browser / Web
- Embedded
- iOS

Image sources:
https://spring.io/
https://medium.com/@afinlay/whats-new-in-java-fx-java-9-updates-a90dd3d4dbba
https://commons.wikimedia.org/wiki/File:Android_robot.svg

Kotlin/Anywhere

Source: https://www.youtube.com/watch?v=3Lqiupxo4CE

Kotlin‘s Goals

- **Design goals**
  - Concise
  - Safe
  - Interoperable
  - Tool-friendly
- **Support large-scale software** (tooling)
- **Mitigate weaknesses from Java**
  - e.g. boilerplate and unsafe arrays
- **Enforce best practices**
  - Immutability, designing for inheritance, …

Get Started With Kotlin

DAY 1

- ✓ Brief Motivation
- ○ **Diving Into Kotlin**
- ○ Functional Programming

DAY 2

- ○ Object-Orientation
- ○ Where to Go From Here

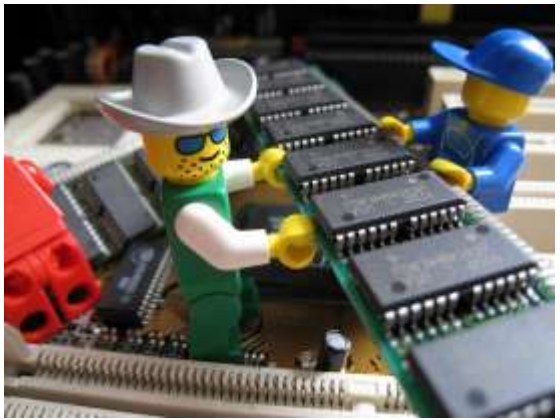
Diving Into Kotlin

1. **Setup**
2. Variables
3. Data Types
4. Collections
5. Control Flow
6. Functions
7. Null Handling
8. Idiomatic Code

Setup

1. Start IntelliJ
2. Clone GitHub repo (see resources)
    1. Or download as ZIP
3. Open folder in IntelliJ
4. (Maybe set up Project SDK)

5. You‘re all set!



http://blog.18004memory.com/2011/04/07/installing-computer-memory/

Diving Into Kotlin

- ✓ Setup
- **2. Variables**
- 3. Data Types
- 4. Collections
- 5. Control Flow
- 6. Functions
- 7. Null Handling
- 8. Idiomatic Code

Variables

- **Focus on immutability**
  - `val` vs `var`
- **Type inference**
  - `val name = "Peter"`
  - What's the type?
  - You can infer, and so should the compiler
- **May be declared on file-level**

Diving Into Kotlin

- ✓ Setup
- ✓ Variables
- **3. Data Types**
- 4. Collections
- 5. Control Flow
- 6. Functions
- 7. Null Handling
- 8. Idiomatic Code

Data Types I – Basics

- **Same basic types as Java**
  - Byte, Short, Int, Long
  - Float, Double
  - Char, String
  - Boolean

Data Types II – Nullables

- **Nullable types**
  - Variables cannot usually be null
  - Must be explicitly nullable
  - Prevents NullPointerException

```
val username: String = "Peter"
val username: String? = null
```

Diving Into Kotlin

- ✓ Setup
- ✓ Variables
- ✓ Data Types
- **4. Collections**
- 5. Control Flow
- 6. Functions
- 7. Null Handling
- 8. Idiomatic Code

Collections

- **Collections**
  - Focus on immutability (precisely: read-only)
  - Lists: `listOf()` vs `mutableListOf()`
  - Set: `setOf()` vs `mutableSetOf()`
  - Map: `mapOf()` vs `mutableMapOf()`
  - Array: `arrayOf()` vs `intArrayOf()` etc

- (Read-only collections are covariant)

Diving Into Kotlin

- ✓ Setup
- ✓ Variables
- ✓ Data Types
- ✓ Collections
- **5. Control Flow**
- 6. Functions
- 7. Null Handling
- 8. Idiomatic Code

Control Flow I – Conditionals

- **Kotlin has `if` and `when`**
  - **`when`** is like Java's **`switch`** but on steroids
- Both are *expressions*
- **`if`** replaces ternary condition operator
  - `val msg = if (hasAccess) hello() else login()`

```
if (isActive) {
  doStuff()
} else {
  cleanup()
}
```

```
when (state) {
  WAITING -> wait()
  RUNNING -> run()
}
```

Control Flow II – Loops

- **The `for`-loop**
  - For any `Iterable`
  - Not `(init; condition; action)` structure!
- **The `while`-loop**
  - Same as in Java and others
- **The `do-while`-loop**
  - Same as in Java and others

```
for (x in -3..3) {
  y[x] = f(x)
}
```

```
while (isActive) {
  keepWorking()
}
```

Diving Into Kotlin

- ✓ Setup
- ✓ Variables
- ✓ Data Types
- ✓ Collections
- ✓ Control Flow
- **6. Functions**
- 7. Null Handling
- 8. Idiomatic Code

Functions I – Basics

- **Declared with `fun`**
  - `fun foo() {}`
- **Types always follow name in Kotlin**
  - `fun foo(): Int {}`
  - `fun bar(n: Int) {}`
  - `fun baz(n: Int): Double {}`

Functions II – Features for Great Good

- **Shorthand syntax**
  - `fun area(radius: Double) = Math.PI * radius * radius`
- **Default parameter values**
  - `fun join(strings: List<String>,`
    `delimiter: String = ", ", prefix = "", postfix = "")`
- **Named parameters**
  - `join(names, postfix = "Students: ")`

Functions III – Extension Functions

- **Enhance third-party API**
  - Instead of `FooUtils` classes
- **Extended class is called *receiver***
  - Instance available as `this`
- **Add extension functions to any class**
  - ```
    fun ViewGroup.inflate(layout: Int): View {
      return LayoutInflater.from(context)
          .inflate(layout, this, false)
    }
    ```
  - `myViewGroup.inflate(R.layout.foo)`
    ```

Functions IV – Infix Functions

- **Syntactic sugar**
  - `"Peter" to 3.9 == to("Peter", 3.9)`
  - `set(SHOW_LABELS to true)`
  - `0 until 9`
- Only for members/extensions with one parameter

```
infix fun String.withPrefix(p: String) = p + this

"World!" withPrefix "Hello, "
```

Functions V – Local Functions

- So far only top-level functions

- **Local functions**
  - Functions inside functions
  - Good practice to minimize scope

```
fun foo() {
  // …
  fun localHelper() { … }
}
```

Functions VI – Operators

- **Operators**: `+, -, *, /, +=, -=, ==, in, .., ++, …`
  - Some in Java: `equals()`, `compareTo()`
  - Additional: `plus()`, `minus()`, `plusAssign()`, `contains()`, …
- Only use for operations that fulfill conventions
  - E.g. + is commutative and adds something

```
fun TimeRange.contains(timestamp: Long) =
    this.begin <= timestamp && timestamp <= this.end
```

Reference: Operators

Operator Symbol	Function
`+, -, *, /, %`	`plus, minus, times, div, rem`
`+=, -=, *=, /=, %=`	`plusAssign, minusAssign, timesAssign, divAssign, remAssign`
`==, !=`	`equals`
`<, <=, >, >=`	`compareTo`
`++, --`	`inc, dec`
`[…], […] = …`	`get, set`
`in, !in`	`contains`
`..`	`rangeTo`
`!, +, -`	`not, unaryPlus, unaryMinus`
`(…)`	`invoke`

Functions VII – Varargs

- **Arbitrary number of arguments**
  - `listOf(1, 2, 3, 4, 5, 6, …)`
  - `fun listOf(vararg elements: Int) { … }`
- **Useful to avoid explicit array or list creation**
  - `foo(listOf(1, 2, 3))`
- **Arrays at runtime**
  - Spread operator: `*array`

Functions – Recap

- Function: `fun foo(param: Type): ReturnType { … }`
- Shorthand: `fun foo(param: Type) = ReturnType()`
- Defaults: `fun foo(param: Type = That()) { … }`
- Named: `foo(param: Type = That()) { … }`
- Extensions: `fun Receiver.foo(param: Type) { … }`
- Infix: `infix fun Receiver.foo(other: Type) = …`
  - Call: `receiver foo other`
- Operators: `operator fun Receiver.plus(o: Type) = …`
  - Call: `receiver + other`
- Varargs: `fun foo(vararg params: Type) { … }`

Diving Into Kotlin

- ✓ Setup
- ✓ Variables
- ✓ Data Types
- ✓ Collections
- ✓ Control Flow
- ✓ Functions
- **7. Null Handling**
- 8. Idiomatic Code

Null Handling I – Nullable Types

- **Every type by default not nullable**
  - `val s: String = null` ✗
- **Explicit nullables**
  - `val s: String? = null` ✓
- **Foo is subtype of Foo?**
- **Best practice is to avoid `null`**

Null Handling II – Call Operators

- **Safe call operator**
  - Propagates `null` if receiver is `null`
  - Cannot cause NPE
  - `nullable?.someMethod()`
- **Elvis operator**
  - Operation for `null` case
  - `nullable?.someMethod() ?: somethingElse()` ("or else")
- **Unsafe call operator**
  - You assure compiler you know variable cannot be `null`!
  - `Nullable!!.someMethod()`

Null Handling III – Mapped Types

- **Kotlin compiles to JVM**
  - Java: only primitives not nullable
- **Basic types map to primitive types**
  - `Int` to `int`, `Double` to `double`, `Char` to `char`, …
- **Nullable types map to classes**
  - `Int?` to `Integer`, `Double?` to `Double`, `Char?` to `Character`

Diving Into Kotlin

- ✓ Setup
- ✓ Variables
- ✓ Data Types
- ✓ Collections
- ✓ Control Flow
- ✓ Functions
- ✓ Null Handling
- **8. Idiomatic Code**

Idiomatic Code

- **Idiomatic = following conventions**
  - Comprehensible and readable

1. Prefer immutability
   1. `val` > `var` and `listOf` > `mutableListOf`
2. Use shorthand for single-expression functions
3. Use conditionals as expressions
4. Enhance APIs with extensions
5. Define operators judiciously
6. Avoid nullability
7. Beware of the unsafe call operator!

Diving Into Kotlin

- ✓ Setup
- ✓ Fundamentals
- ✓ Variables
- ✓ Data Types
- ✓ Collections
- ✓ Control Flow
- ✓ Functions
- ✓ Null Handling
- ✓ Idiomatic Code

Image source:
http://www.adriansautos.com.au/services

Get Started With Kotlin

DAY 1

- ✓ Brief Motivation
- ✓ Diving Into Kotlin
- ○ Functional Programming

DAY 2

- ○ Object-Orientation
- ○ Where to Go From Here

Get Started With Kotlin

DAY 1

- ✓ Brief Motivation
- ✓ Diving Into Kotlin
- ○ **Functional Programming**

DAY 2

- ○ Object-Orientation
- ○ Where to Go From Here

Functional Programming

1. Idea and Concepts
2. Function types
3. Higher-order functions
4. Lambdas
5. Collections
6. Scope functions
7. Lazy sequences
8. Inline functions

Functional Programming I – Overview

- **Paradigm like OO**
  - Focus on functions, not objects
- **Functions as first-class citizens**
  - Store in variables
  - Pass around
- **Program = composition of functions**
  - Modularity

Functional Programming II – Benefits

- **Concise & expressive**
  - e.g. working with collections
- **Performance**
  - Lazy evaluation
  - Inlining
- **Simpler solutions**
  - New kinds of modularity
  - e.g. higher-order functions
- **New possibilities**
  - Infinite data structures

Functional Programming

1. ✓ Idea and Concepts
2. Function types
3. Higher-order functions
4. Lambdas
5. Collections
6. Scope functions
7. Lazy sequences
8. Inline functions

Function Types

- **Kotlin has proper function types**
  - `val timesTwo: (Int) -> Int = …`
  - Function from Int to Int
  - Functions are just objects

- **Examples**
  - `(String) -> Int`
  - `(Long) -> Unit`
  - `() -> Int`
  - `(Int, () -> Unit) -> (() -> Unit)`

Functional Programming

- ✓ Idea and Concepts
- ✓ Function types
3. Higher-order functions
4. Lambdas
5. Collections
6. Scope functions
7. Lazy sequences
8. Inline functions

Higher-Order Functions (HOF) I – Basics

- **Higher-order functions**
  - Function as parameter
  - Function as return type
- **Enable new kinds of modularity**
  - e.g. Strategy pattern
- **Extremely useful and thus fundamental concept**
  - Used throughout standard library

Functional Programming

- ✓ Idea and Concepts
- ✓ Function types
- ✓ Higher-order functions
4. Lambdas
5. Collections
6. Scope functions
7. Lazy sequences
8. Inline functions

Lambda Expressions

- **Lambda = anonymous function**
  - Inline function
  - Useful if used in one place only

- **Easy way to denote a function ad-hoc**
  - Powerful in combination with HOF

- **Examples**
  - `{ x: Int, y: Int -> x + y }`
  - `{ times: Int, op: () -> Int -> repeat(times, op) }`
  - Typically types inferred

Functional Programming

- ✓ Idea and Concepts
- ✓ Function types
- ✓ Higher-order functions
- ✓ Lambdas
5. Collections
6. Scope functions
7. Lazy sequences
8. Inline functions

Collections I – Useful Functions

- **Powerful standard library for collections**
  - filter
  - map
  - zip
  - fold
  - reduce
- **Can be chained**
  - Goal: concise but readable

Functional Programming

- ✓ Idea and Concepts
- ✓ Function types
- ✓ Higher-order functions
- ✓ Lambdas
- ✓ Collections
6. Scope functions
7. Lazy sequences
8. Inline functions

Scope Functions – Overview

- **Higher-order extensions from stdlib**
  - Useful for scoping, nullables and more
  - Used frequently in idiomatic code
  - Takes getting used to

- Five main functions
  - `let`
  - `with`
  - `apply`
  - `also`
  - `run`

Reference: Scope Functions

- **`let`:** nullables or scoping
  - `nullable?.let { doOnlyIfNotNull() }`
  - `File("…").reader().let { … }`
- **`with`**: many calls on one object or access extensions
- **`apply`**: initialization or builder-style
  - `Training().apply { topic = "Kotlin"; hours = 8 }`
- **`also`**: actions on the side or validation
  - `doSomething().also { require(…); log(…) }`
- **`run`**: open new scope or immediately apply function
- **(`use`**: try-with-resources)

Functional Programming

- ✓ Idea and Concepts
- ✓ Function types
- ✓ Higher-order functions
- ✓ Lambdas
- ✓ Collections
- ✓ Scope functions
7. Lazy sequences
8. Inline functions

Lazy Sequences I – Overview

- **Eager evaluation vs lazy evaluation**
  - Eager: evaluate all expressions immediately
  - Lazy: evaluate only on demand, and only what's necessary
  - `words.filter { "z" in it ) }.map { it.reversed() }.take(2)`

- **Allows infinite sequences**
  - New values computed on demand (lazily)

- **May improve performance**
  - If many elements and expensive computation steps
  - No intermediate objects

Lazy Sequences II – Intuition

- **Similar to lists but lazy**
  - Represents multiple elements
- **Pretty much like Java 8 Streams**
  - But compatible with Java 6
  - Android
- **Transformations vs actions**
  - Transformations define evaluation steps
  - Only actions trigger actual evaluation

Functional Programming

- ✓ Idea and Concepts
- ✓ Function types
- ✓ Higher-order functions
- ✓ Lambdas
- ✓ Collections
- ✓ Scope functions
- ✓ Lazy sequences
- 8. Inline functions

Inline Functions I – Idea

- **Each HOF is object + has closure**
  - Requires memory
  - Can be mitigated by inlining the HOF
  - Lambda parameters inlined as well
  - No anonymous classes created

- **Use in...**
  - Small functions (beware of bytecode size)
  - Loop call-sites (pays multifold)

Inline Functions II – Usage

- **Inline modifier**
  - `inline fun lock(lock: Lock, op: () -> Unit) { … }`

- Inlined result of `lock(myLock) { foo() }`:

```
myLock.lock()
try {
  foo()
} finally {
  myLock.unlock()
}
```

Inline Functions III – Notes

- **Exclude lambda parameters from inlining**
  - `inline fun foo(noinline param: () -> Unit) { … }`

- **Inlining enables non-local returns**
  - Normal lambdas cannot `return` (without label)
  - Imagine the `return` in the code after inlining
    - Works as usual

Functional Programming

- ✓ Idea and Concepts
- ✓ Function types
- ✓ Higher-order functions
- ✓ Lambdas
- ✓ Collections
- ✓ Scope functions
- ✓ Lazy sequences
- ✓ Inline functions

Image source:
http://www.adriansautos.com.au/services

Get Started With Kotlin

DAY 1

- ✓ Brief Motivation
- ✓ Diving Into Kotlin
- ✓ Functional Programming

DAY 2

- ○ Object-Orientation
- ○ Where to Go From Here