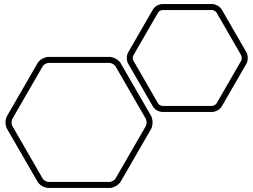
SOURCELABS

Kotlin for Spring Developers



Practical information

Morning

More than an introduction to Kotlin

Lunch

~60 minutes

Afternoon

What Spring has to offer to Kotlin developers

About me

- **Stephan Oudmaijer**
- Started Sourcelabs in 2017
- Programming Java since version 1.1
- Tried other languages like VB, PHP, Ruby, C++, JavaScript, Scala, Groovy
- Like to work with new technologies and to figure out if it brings value
- Hobbies: CrossFit and windsurfing

Raise hands

- Who is longer than 5 years in software development?
- Who (still) likes to write Java code?
- Who has already worked with Kotlin?
- Who has experience with Maven projects?
- Who has more than 5 years experience with Spring Boot?

Kotlin timeline

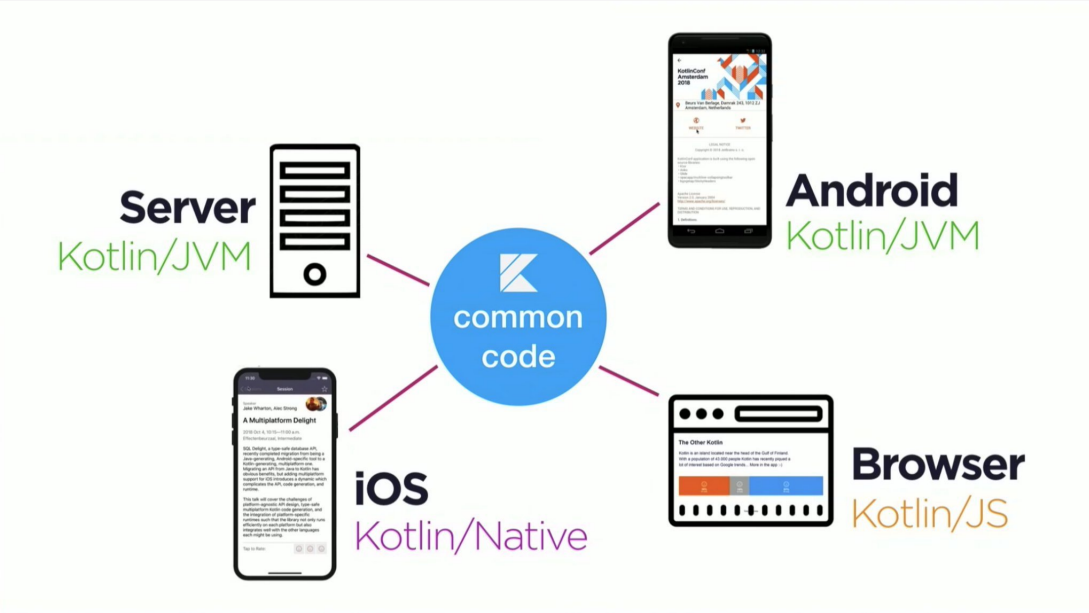
- 2010 Project started
- 2011 Project revealed
- 2012 Open-source
- 2014 Koans
- 2016 Kotlin 1.0
- 2017 Official on Android
- 2017 Kotlin support Spring 5
- 2018 Kotlin 1.3
- 2020 Kotlin 1.4

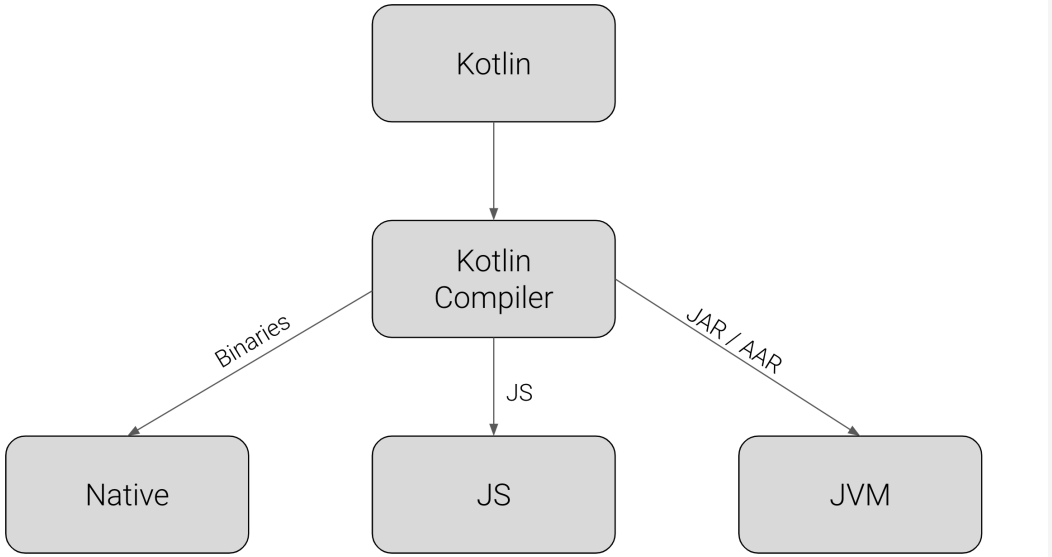
Kotlin

- General purpose language
  - Supports OO and functional programming paradigms
- Staticly typed
  - Although types can be mostly omitted
- Pragmatic
  - Not a research project
- Open-source project
  - Mainly developed by JetBrains

Kotlin Java Interoperability

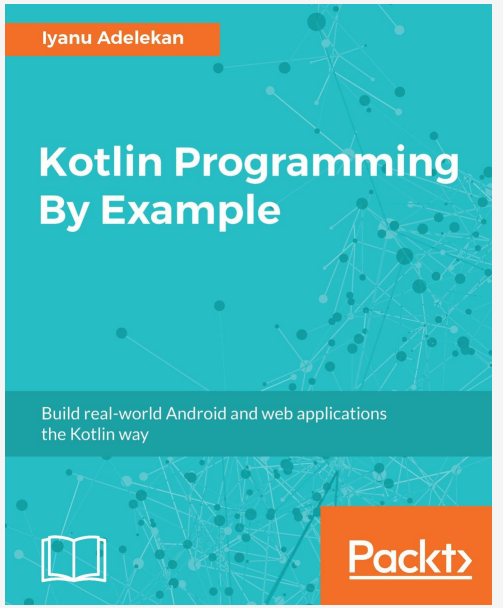
Kotlin = multiplaform

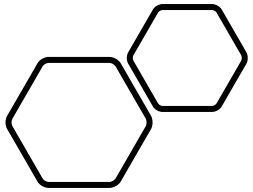




Learning Kotlin

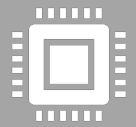
- Kotlin Koans
  - https://play.kotlinlang.org/koans/overview
- Coursera
  - https://www.coursera.org/learn/kotlin-for-java-developers
- Kotlinlang
  - https://kotlinlang.org/





How to start with Kotlin?

Gain some real-world experience with Kotlin



Introduce in parts of your existing (production) applications

Whats next?

- Watch some videos
- Do some exercises

From Java to Kotlin

- Video: 01. From Java to Kotlin

Converted Kotlin code != Idiomatic Kotlin code

Writing **idiomatic Kotlin** code is **manual work**

Converting Java code **to** Kotlin results in **Java style Kotlin code**

Hello, World in Kotlin

- Video: 02. Hello, World

Getting started with Kotlin in your (Maven) project

- Configuring Kotlin in Maven (in IntelliJ)
- Convert Java code to Kotlin using IntelliJ

Maven: Kotlin dependencies (pom.xml)

```
<project xmlns="http://maven.apache.org/POM/4.0.0" ...>

<properties>
   <kotlin.version>1.3.70</kotlin.version>
</properties>

<dependencies>
   <dependency>
      <groupId>org.jetbrains.kotlin</groupId>
      <artifactId>kotlin-stdlib</artifactId>
      <version>${kotlin.version}</version>
   </dependency>
</dependencies>

<build>
   <plugins>
      <plugin>
         <groupId>org.jetbrains.kotlin</groupId>
         <artifactId>kotlin-maven-plugin</artifactId>
         <version>${kotlin.version}</version>

...
```

Maven: configuring the Kotlin compiler (pom.xml)

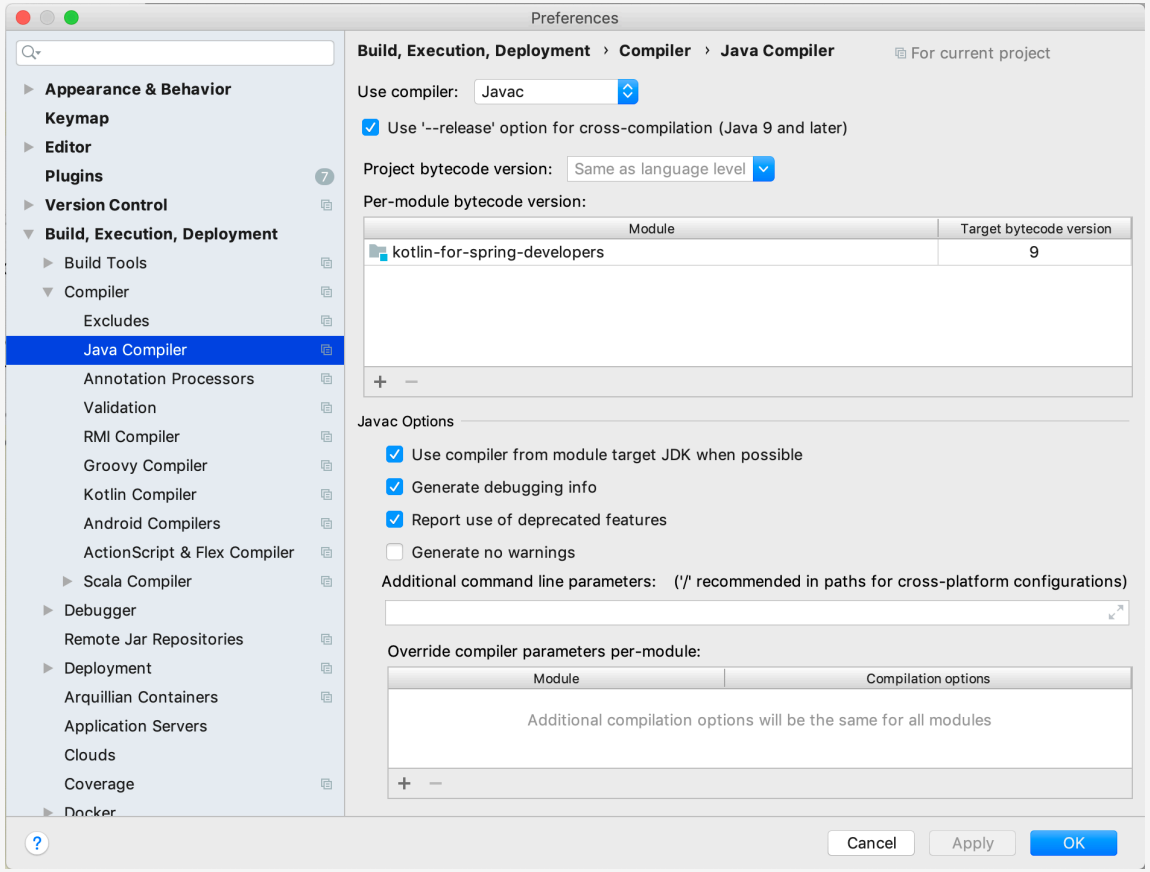
Java compiler

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>11</source>
    <target>11</target>
  </configuration>
</plugin>
```

Kotlin compiler

```
<plugin>
  <groupId>org.jetbrains.kotlin</groupId>
  <artifactId>kotlin-maven-plugin</artifactId>
  <executions>
    <execution>
      <id>compile</id>
      <phase>compile</phase>
      <goals>
        <goal>compile</goal>
      </goals>
    </execution>
    <execution>
      <id>test-compile</id>
      <phase>test-compile</phase>
      <goals>
        <goal>test-compile</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <jvmTarget>11</jvmTarget>
  </configuration>
</plugin>
```

Mind the IntelliJ compiler settings!



Demo!

- Configuring a Maven project with Kotlin in IntelliJ
- Convert *Hello.java* to Kotlin

File: *Hello.java*

```
package exercise;

public class Hello {

    public static void main(String[] args) {
        String name = "Java";
        System.out.println("Hello, " + name + "!");
    }
}
```

Exercise: getting started with Kotlin

- Create a new Maven project in IntelliJ
- Create the file *Hello.java*
- Convert *Hello.java* to Kotlin
- Configure Kotlin in Maven

File: *Hello.java*

```
package exercise;

public class Hello {

    public static void main(String[] args) {
        String name = "Java";
        System.out.println("Hello, " + name + "!");
    }
}
```

Cheat sheet: https://github.com/sourcelabs-nl/kotlin-for-spring-developers

Variables

- Video: 03. Variables

Variables

- var: mutable
- val: read-only (after assignment) like **final** in Java
- Mutability also explicit for collection types
- Local type inferred by compiler

Exercise: convert to Kotlin - variables

File: *Variables.java*

```
package exercise;

import java.util.*;

public class Variables {

    public static void main(String[] args) {
        String mutableString = "string1";
        mutableString = "mutated";
        final String immutableString = "string2";
        List<String> mutableList = Collections.unmodifiableList(new ArrayList<String>());
        List<String> immutableList = new ArrayList<>();
    }
}
```

Functions

- Video: 04. Functions

Named & default arguments

- Video: 05. Named & default arguments

Functions in Kotlin

- Function **body** can be an **expression**
- Different types: **top-level**, **member** and **local** function
- Top-level function are like static functions in Java
- Function **arguments** can be called **by name**
- Function **arguments** can have **default values**

Exercise: functions & arguments - implement in Kotlin

- Create *Calculator.java*
- Implement *Calculator* in Kotlin
- Make sum an expression body

File: *Calculator.java*

```
package exercise;

public class Calculator {

    public int sum(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        Calculator f = new Calculator();
        System.out.println(f.sum(0, 1));
    }
}
```

Conditionals if & when

- Video: 6. Conditionals if & when

Conditionals if & when

- **if** and **when** are expressions, you can assign the result to a variable
- No ternary operator in Kotlin
- Smart casts in if statement by using the **is** keyword

Exercise: if & when

- Create *IfWhen.java*
- Implement *IfWhen* in Kotlin
  - Write top-level functions
  - *Any* is *Object* in Kotlin
  - Cant use *tenary* operator
  - Use *is* instead of *instanceof*
  - Use *when* in *yesNoToBoolean()*

File: *IfWhen.java*

```
package exercise;

public class IfWhen {

    public static boolean isString(Object o) {
        return o instanceof String ? true : false;
    }

    public static boolean yesNoToBoolean(String s) {
        if ("yes".equals(s))  return true;
        else if ("no".equals(s)) return false;
        else throw new RuntimeException("Unsupported value: " + s);
    }

    public static void main(String[] args) {
        System.out.println(isString(Integer.MAX_VALUE));
        System.out.println(yesNoToBoolean("yes"));
    }
}
```

Loops

- Video: 07. Loops

Loops

- Iterate using **in** keyword
- Use **.withIndex** for a for-loop with an **(index,element)** variable

Exercise: loops

- Create *Loops.java*
- Implement *Loops* in Kotlin
  - Iterate the loop using the *in* keyword
  - Use the ranges syntax
    - **for** (i **in** 0..9)

File: *Loops.java*

```
package exercise;

import java.util.List;

public class Loops {
    public static void main(String[] args) {
        List<String> list = List.of("a", "b");

        for (String s : list) {
            System.out.println(s);
        }

        for (int i=0; i<10; i++) {
            System.out.println("i="+ i);
        }
    }
}
```

Exceptions

- Video: 09. Exceptions

Exceptions

- Kotlin has no checked exceptions
- **throw** and **try** are expressions: you can return or assign the result to a variable
- **@Throws** for Java interop

Exercise: handling exceptions

- Create *FileReader.Java*
- Run the *main* function
- Create *FileReaderKotlin.kt*
- Invoke FileReaderKotlin
- **new** FileReaderKotlin()
  .readBytes("**nonexisting**");
- Catch the IOException

File: *FileReader.java*

```
package exercise;

public class FileReader {

    public byte[] readBytes(String filename) throws IOException {
        return Files.readAllBytes(Paths.get(filename));
    }

    public static void main(String[] args) {
        // TODO catch the IOException
        new FileReader ().readBytes("nonexisting");
    }
}
```

File: *FileReaderKotlin.kt*

```
package exercise

class FileReaderKotlin {
    fun readBytes(filename: String) = File(filename).readBytes()
}
```

Extension functions

- Video: 10. Extension functions

Std lib extensions

- Video: 11. Std lib extensions

Extension functions

- An extension function extends a **Class**
- The **type** is called the **receiver**
- Inside extension we can access the receiver by **this** or can be **omitted**
- Extensions need be **imported** for usage
- From Java a regular **static** function with receiver as first argument
- Typically defined as top-level functions in separate file

Std lib extension

- Kotlin std lib = Java standard library + extensions
- Spring Kotlin support = Spring Java libraries + extensions
- No Kotlin SDK but small runtime jar

Exercise: create an extension function

- Create *KotlinExtensions.kt*
- Create the *main* function
- Implement *countUpperCase()*

File *KotlinExtensions.kt*

```
fun main() {
    println("abCdE".countUpperCase())
}
```

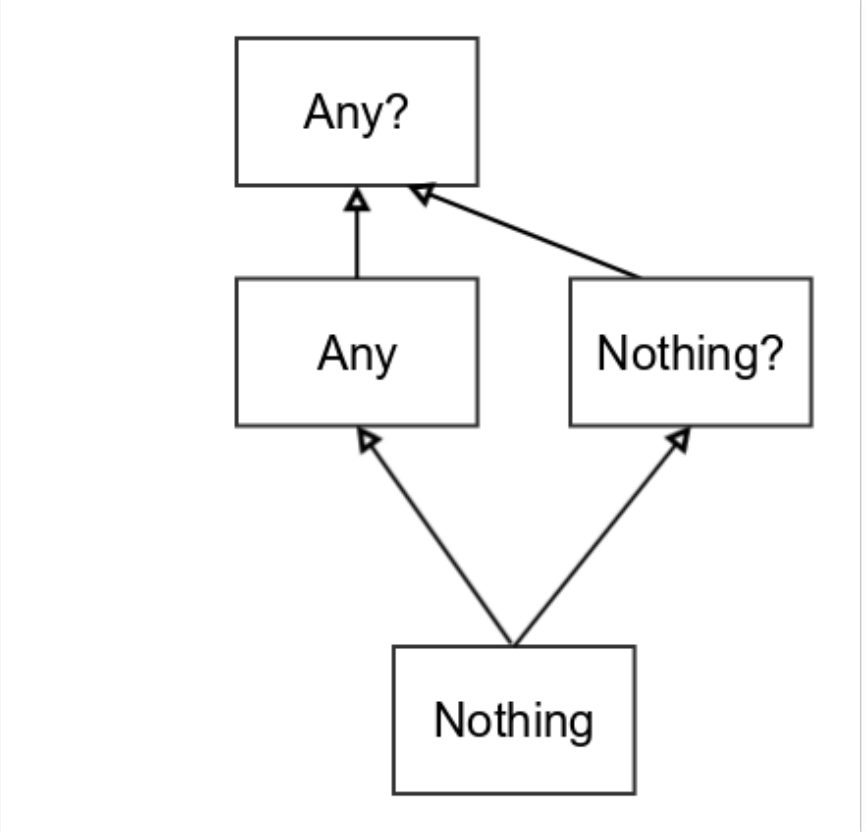
Nullable types

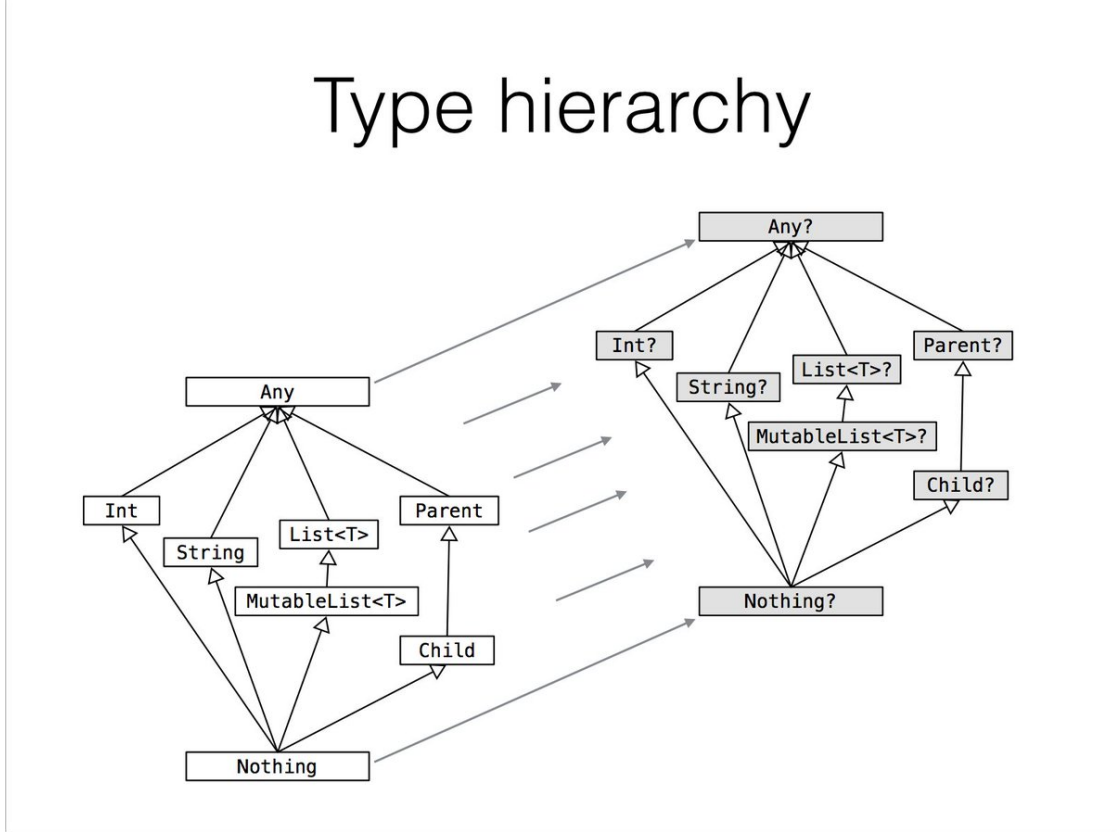
- Video: 14. Nullable types

Nullable types

- Nullability should be compile time, not a runtime problem!
- Types can be either nullable or non nullable
- Smart cast applied by compiler to non nullable type
- Safe access operation **?.**
- Default value using elvis operator **?:**
- Not null assertion operator **!!**
- Be aware of operator precedence

Kotlin type hierarchy





Kotlin type hierarchy

kotlin-stdlib / kotlin / Nothing

Nothing

```
class Nothing
```

Nothing has no instances. You can use Nothing to represent "a value that never exists": for example, if a function has the return type of Nothing, it means that it never returns (always throws an exception).

Nullable types under the hood

- Video: 15. Nullable types under the hood

Nullable types under the hood

- Kotlin adds @Nullable and @NotNull annotations
- Compiler checks annotations during compilation
- Nullable types != Optional
- No overhead (like Optional wrappers)
- Nullability also reflected in generics

Exercise: Nullability

Create *Nullability.kt*

Fix the code

```
package exercise

import org.jetbrains.annotations.NotNull

class Nullability {

    fun fixme(@NotNull str: String?) {
        val length = str.length
        print("$str length is $length")
    }
}
```

Lambdas

- Video: 18. Lambdas

Common operations on collections

- Video: 19. Common operations on collections

Lambdas

- Lambda is an **anonymous function** that can be **used as an expression**
- Defined in curly braces **{ }**
- Lambda syntax: **{** arguments **->** body **}**
- When lambda is last or only argument of a function: can be moved outside arguments
- Implicit name of single parameter **it**

Lambdas

- Example Lambda to filter items in a list

Filter:

```
listOf("a", "b").filter( { item -> item == "a" })
listOf("a", "b").filter() { item -> item == "a" }      // Moved out of parentesis
listOf("a", "b").filter   { item -> item == "a" }      // Omit parentesis
listOf("a", "b").filter   { it == "a" }                // Implicit receiver it
```

Common operations on collections

- Kotlin alternative for Java Stream API using Lambda syntax
- Commonly used: *filter*, *map*, *any*, *none*, *all*, *first*, *firstOrNull*, *find*, *count*, etc.

Exercise: common operations on collections

Create *Collections.kt*

Print the drinks without alcohol

Print the number of drinks with alcohol

```
package exercise

class Drink(val name: String, val alcohol: Int)

fun main() {
    val drinks = listOf(
            Drink("Coca-Cola", 0),
            Drink("IceTea", 0),
            Drink("Heineken", 5)
    )
    // TODO
}
```

Object oriented programming in Kotlin

- Video: 23. OOP in Kotlin

Constructors and inheritance

- Video: 24. Constructors, Inheritance syntax

Class modifiers

- Video: 25. Class modifiers

Object oriented programming in Kotlin

- Default visibility **public final**
  - This breaks for example JPA or CGLIB enhancement!
- **open** makes a class or function non-final
- **internal** for module accessibility (kotlin only)
- Package structure different
  - Multiple classes allowed in a single file
  - Package name does not need to represent directory structure

Constructors and inheritance

- No **new** keyword to create an instance of a class
- init {} represents the constructor body
- Primary and secundairy constructors
- Constructor keyword can be omitted but might be useful for annotations

```
class Person constructor(val name: String)
```

```
class Person (val name: String)
```

Class modifiers

- **enum** for enum Classes
- **data** keyword adds *equals*, *hashCode*, *toString*, *copy* functions to a class

Exercise: class & data class

Create *Address.java*

Convert it to a Kotlin class

Convert it to a Kotlin **data** class

Create a main function

Create 2 address instances

Compare the 2 addresses

- By zipCode and houseNumber

```
public class Address {
    private final String zipCode;
    private final int houseNumber;
    private String street;

    public Address(String zipCode, int houseNumber) {
        this(zipCode, houseNumber, null);
    }

    public Address(String zipCode, int houseNumber, String street) {
        this.zipCode = zipCode;
        this.houseNumber = houseNumber;
        this.street = street;
    }

}
```

Generics

- Video: 29. Generics

Generics

- Result can be nullable: T?
- Makes the input type explicit non nullable: fun <T : Any>
- Use @JvmName on function with different generified types

Library functions looking like built-in constructs

- Video: 30. Library functions looking like built-in constructs

More useful library functions

- Video: 32. More useful library functions

Library functions looking like built-in constructs

- Functions like: run, let, takeIf, takeUnless, repeat
- Using lambda with receivers: with, run, apply

Exercise: common library functions

Create *Address.java*

Create *UseAddress.kt*

Write a *main* function

Construct an Address

- Using *apply*: Address().*apply* **{ … }**
- Try the same using *with*(address) **{ … }**

```
package exercise;

public class Address {
    private String zipCode;
    private int houseNumber;

    public String getZipCode() { return zipCode; }

    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }

    public int getHouseNumber() { return houseNumber; }

    public void setHouseNumber(int houseNumber) {
        this.houseNumber = houseNumber;
    }
}
```

Lambda with receivers

- Video: 31. Lambda with receiver

Lambda with receivers

- Also called extension lambdas
- Ideal for writing DSLs
- Can remove boilder-plate like instantiating objects

Example: built-in std library functions

```
val sb = StringBuilder()

// public inline fun <T, R> T.run(block: T.() -> R): R
sb.run {
    append("Hello,")
    append("Kotlin!")
    toString()
}

// public inline fun <T, R> with(receiver: T, block: T.() -> R): R
with(sb) {
    append("Hello,")
    append("Kotlin!")
    toString()
}
```

Lambda with receivers (built-in)

```
buildString {
    append("Hello,")
    append("Kotlin!")
}
```

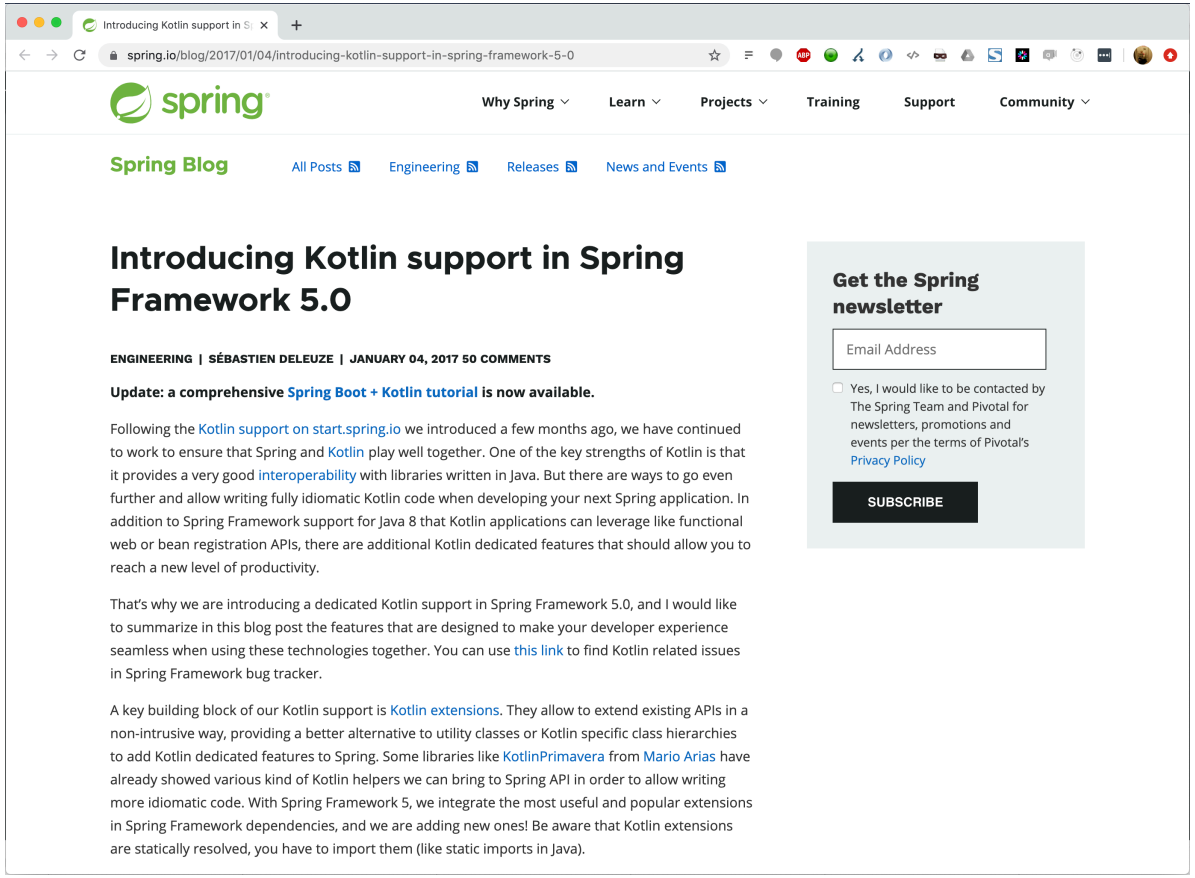
```
public inline fun buildString(builderAction: StringBuilder.() -> Unit): String {
    val builder = StringBuilder()
    builder.builderAction()
    return builder.toString()
}
```

```
public inline fun buildString(builderAction: StringBuilder.() -> Unit): String =
        StringBuilder().apply(builderAction).toString()
```

Exercise: write a StringBuilder using Lambdas with receiver

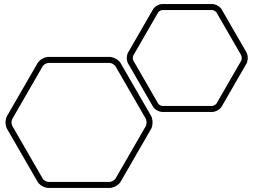
```
myStringBuilder {
    append("Hello,")
    append("Kotlin!")
}
```

```
public inline fun myStringBuilder(builderAction: StringBuilder.() -> Unit): String {
    val builder = …
    …
    return builder.toString()
}
```



Kotlin support in Spring

- Since Spring 5 (2017)
- Sébastien Deleuze
- Spring Java libraries + annotations + extensions
- Added @NotNull and @Nullable to Spring code base
- Bean Definition DSL
- Spring Boot Coroutines support



Workshop: Kotlin Bootique

Use what you have learned and apply it to an existing Spring Boot Java project

Follow the guide on github

https://github.com/sourcelabs-nl/kotlin-bootique/blob/master/README.md

Thats all!

- Next steps
  - You will receive the slides
  - And an evaluation form (anonymous)

- Thanks for attending!!

Bonus

- Next >>

Objects, object expressions and companion objects

- Video: 27. Objects, object expressions and companion objects

Objects, object expressions and companion objects

- Object is same as a singleton pattern in Java
  - Special INSTANCE variable available
- Object expression: object : SomeClassToOverride() {}
- Kotlin does not have static methods, @JvmStatic for interop
- Classes van have companion object
  - Companion object can implement an interface
  - Accessible via Companion instance

Function types

- Video: 20. Function types

Function types

- Function type is a stored lambda in a variable
- Type declaration: () -> T
  - () defines input params
  - -> T the return type
- Can be used with SAM (single abstract method) classes
  - val runnable = *Runnable* { println(42) }
  - val rowMapper = *RowMapper<Person>* { rs, index -> Person(rs.getString("name")) }

Exercise: write a lambda function type

- Define a function type for a lambda that adds two numbers and returns the result
- Create the lambda expression
- Write a main and Invoke it

- Example: val x: (p1, p2) -> T = **{** p1, p2 -> body **}**
  - () defines input params
  - -> T the return type
  - **{}** defines the lambda expression

Exercise: lambda with receiver

```
class URL(val host: String, val port: Int) {

    class Builder {
        var host: String = "localhost"
        var port: Int = 80
        fun build(): URL = URL(host, port)
    }
}

fun main() {
    val builder = URL.Builder()
    builder.host = "www.nu.nl"
    builder.port = 8080
    builder.build()
}
```

```
fun url(function: URL.Builder.() -> Unit): URL {

        // TODO

}
```

```
fun main() {
    url {
        host = "www.nu.nl"
        port = 8080
    }
}
```