Introduction to Kotlin

Andy Bowes

The JVM Thing

20th July 2017

Aim of Presentation

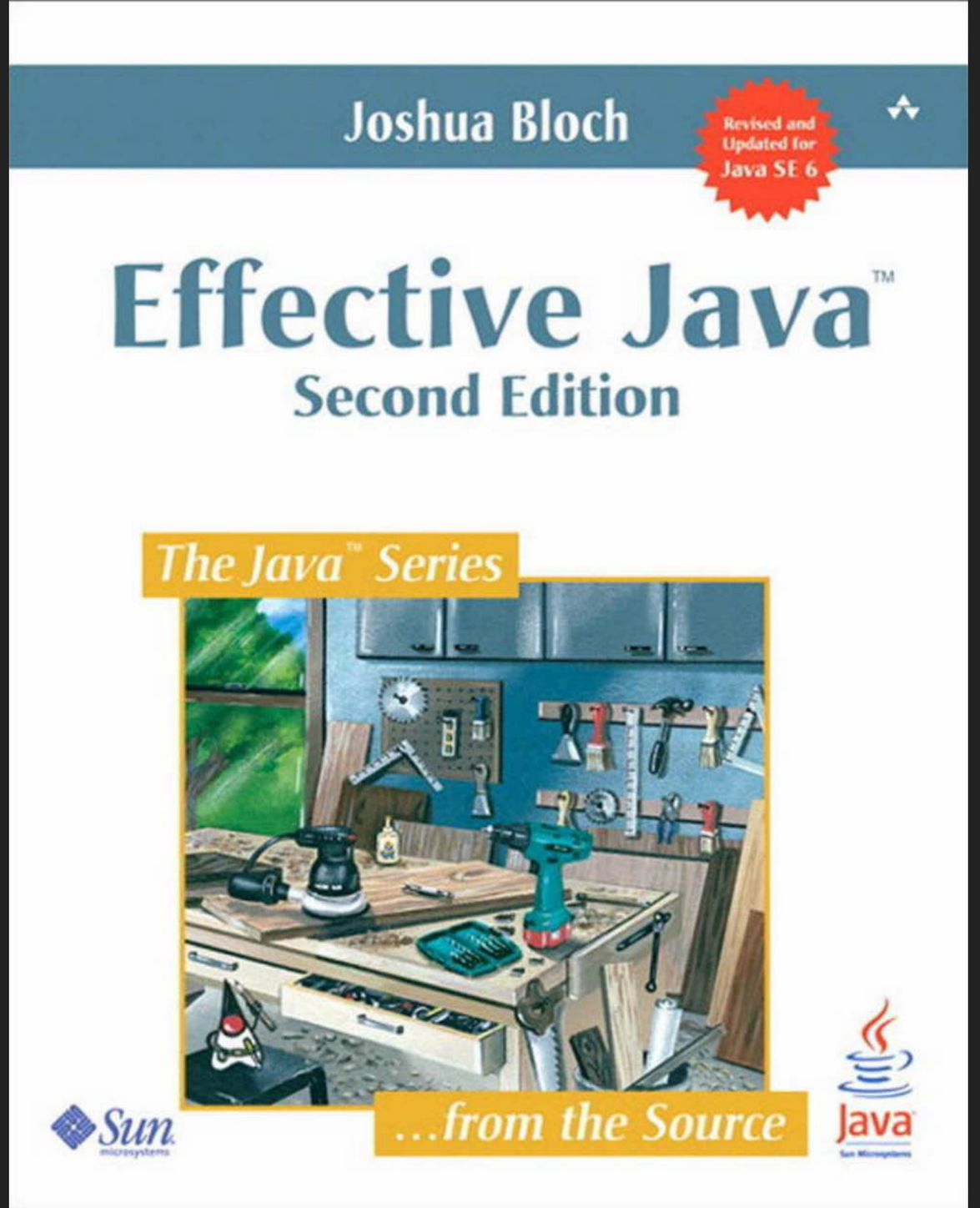
- Provide Overview of Kotlin
- Origin of Kotlin
- Introduce basic syntax
- Provide a 'feel' for the language

What is ‘Kotlin’?

- ‘New’ programming language
  - Developed since 2011, v1.0 released Feb 2016
- Developed by JetBrains
  - Makers of IntelliJ & Android Studio
- Runs on Java Virtual Machine (JVM)
  - JavaScript
  - Native
- Open Source

What's wrong with plain old Java ?

- It's verbose.
  - Too much 'boilerplate' code.
- Slow to Change
- Multiple overload methods/constructors.
- Null Pointers
- Class Cast exceptions
- Functional paradigm is still a bit of an afterthought.


Joshua Bloch
Revised and Updated for Java SE 6
Effective Java™
Second Edition
The Java™ Series
...from the Source
Sun microsystems
Java

James Gosling, inventor of Java:

I sure wish I had this book ten years ago. Some might think that I don't need any Java books, but I need this one.

Effective ~~Java~~ Kotlin

- Kotlin is designed as a 'better' Java
- Directly addresses many issues from Joshua Bloch's Effective Java
- Many of these will be highlighted during this talk

Kotlin's Relationship to Java

- All code compiles to pure Java byte-code
- 100% Java Interoperability
  - Kotlin classes can invoke methods in Java classes
  - Java classes can invoke Kotlin functions
- Kotlin can use standard Java libraries
- Allows incremental migration to Kotlin from Java
- Deploy mixed applications as a single artifact

2017 - Leeds JVM Thing, Kotlin Evening

Basic Syntax

Pragmatic language

- Started as JetBrains internal language
- Kotlin is statically typed
- Combination of OO & Functional
- Prefers code clarity over brevity
- 'Borrows' concepts from multiple sources
  - Elements from Groovy, Scala, Ceylon, Swift, ...

Objects Everywhere

- Everything is an object.
- No 'primitive' types.
- Data types are very similar to Java
  - Root class is Any not Object
  - Nothing - null values
  - Unit - void return type

Basic Data Types

- Numeric Types
  - Integer : Long, Int, Short, Byte
  - Decimal : Float, Double
- Text Types
  - Char, String

Variable Definitions

- Reverse of Java notation
  - variable name followed by type
- var - mutable variable
- val - immutable property

```
val customerName : String = "Andy Bowes"
val customerName = "Andy Bowes"
```

Numeric Ranges

```
for (i in 1..100) { ... }
for (i in 1 until 100) { ... }
for (x in 2..10 step 2) { ... }
for (x in 10 downTo 1) { ... }
if (x in 1..10) { ... }
```

Raw Strings

- Raw Strings is deliminated by a triple quote
  - contains no character escaping
  - may contain new lines and other characters

```
val text = """
    for (c in "foo")
        print(c)
"""
```

String Interpolation

- Strings can contain placeholders for variables or expressions
- Cleaner alternative to String.format()

```
val i = 10
val s = "i = $i"

val name = "abc"
val str = "$name.length is ${name.length}"
```

Function Definitions

```
fun getFullName( cust: Customer): String = {
  return "${cust.forename} ${cust.surname}"
}

fun getFullName( cust: Customer): String =
                "${cust.forename} ${cust.surname}"

fun getFullName( cust: Customer) = "${cust.forename} ${cust.surname}"
```

Function Parameters, Default Values

- Ability to define default values for optional parameters
- Reduces need for overloaded methods

```
fun createBook(title: String,
               subtitle : String? = None,
               paperback : Boolean = true,
               price: BigDecimal = BigDecimal(7.99)) : Book{
        ...
}
```

Effective Java: Use Overloading Judiciously

Using Named Parameters

- Ability to specify optional parameters by name
- Overrides default value if supplied

```
createBook("Kotlin in Action",
           paperback=False,
           price = BigDecimal(14.99))
```

Extension Functions

- Add functions onto existing classes
- No need to create a sub-classes

```
fun <T> MutableList<T>.swap(index1: Int, index2: Int){
  val tmp : T = this[index1]
  this[index1] = this[index2]
  this[index2] = tmp
}
```

Using Extension Methods

```
fun String.toCamelCase() : String {
   return this.split(' ').map { it -> it.toLowerCase()
.capitalize() }
.joinToString(separator = "")
}

fun main(args: Array<String>) {
   println("this is a test".toCamelCase())
   println("ANOTHer Test Case".toCamelCase())
}
```

2017 - Leeds JVM Thing, Kotlin Evening

Classes

Classes

- Similar to Java class definition
- Multiple classes can be defined in the same file
- File name is independent of class name
- Default visibility is public
- Final by default

Primary Constructors

```
class Customer(name: String) {
    init {
        logger.info("Customer initialized with value ${name}")
    }
}

val customer = Customer("Guy Edwards")
```

- No code in primary constructor body
- Code placed in optional init blocks
- Instances created without new keyword

Constructors - Default Values

- Constructor arguments can have default values
- Reduces need for overloaded constructors

```
DummyHashMap(val initialCapacity: Int = 16,
             val loadFactor : Float = 0.75F)
```

Secondary Constructors

- Class may have secondary constructor
- Each secondary constructor *must* invoke primary constructor

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Inheritance

- All classes are final unless explicitly declared otherwise
- Overridable methods must be declared open.
- Overrided methods must be declared.

```
open class Base(p: Int){
  fun deleteName(name: String){ ... }
  open fun addName(name: String){...}
}

class Derived(p: Int) : Base(p) {
  override fun addName(name:String) {...}
}
```

Effective Java: Design and document for inheritance or else prohibit it

Effective Java: Consistently use the override annotation

Data Classes - Kotlin

```
import java.util.Date

data class Person(val id: String,
    val forename: String,
    val surname: String,
    val dateOfBirth: Date)
```

Data Classes

Methods for Nothing

Implementations of toString(), equals, hashCode() and copy() are automatically created. Getters created for all properties, and Setters only created for mutable properties.

```
person.equals()
person.hashCode()
person.toString()
person.copy()
```

Effective Java: Always override hashCode when you override equals

Effective Java: Make defensive copies when needed

Implementing Singletons

- Kotlin uses Objects to create Singletons.
- These are convenient placeholders for the equivalent of static Java methods.

```
object DataProviderManager {
    fun registerDataProvider(provider: DataProvider) {
        // ...
    }

    val allDataProviders: Collection<DataProvider>
        get() = // ...
}
```

Effective Java: Enforce the singleton property with a private constructor or an enum type

Companion Objects

- Provide similar functionality to Java static methods

```
class MyClass {
    companion object Factory {
        fun create(): MyClass = MyClass()
    }
}

val myInstance = MyClass.create()
```

Sealed Classes

- Conceptually similar to Enums

```
sealed class ChessPiece
class King() : ChessPiece()
class Queen() : ChessPiece()
class Rook() : ChessPiece()
class Bishop() : ChessPiece()
class Knight() : ChessPiece()
class Pawn() : ChessPiece()

fun move(piece: ChessPiece): Unit = when(piece) {
    is King -> ...
    is Queen -> ...
    is Rook -> ...
    is Bishop -> ...
    is Knight -> ...
    is Pawn -> ...
}
```

Type Aliases

- Added in Kotlin 1.1
- Provides a alias for existing types
- typealias MovieCast = List<Actor>
- Can improve code readability and maintainability

Delegation

Joshua Bloch, Effective Java:

> *Inheritance is a powerful way to achieve code reuse, but it is not always the best tool for the job. Used inappropriately, it leads to fragile software.*

Effective Java: Favour Composition over Inheritance

Delegation Example

```
interface Printable {
    fun print(message: String)
}

class PrintableImpl(val copies: Int) : Printable {
    override fun print(message: String) { for (i in 1..copies) { ... } }
}

class Derived(b: Printable) : Printable by b

val printer = PrintableImpl(copies=2)
val derived = Derived(printer)

derived.print("Test Message")
```

2017 - Leeds JVM Thing, Kotlin Evening

Null Handling

Null Safety

Effective Java: Use assertions to check for Null parameters

- No more Null Pointer Exceptions
- Need to explicitly state that variable/parameter allows nulls
  - val name: String - Cannot be assigned null
  - val name: String? - Can be set to a value or null
- Unsafe calls are prevented by the compiler

Compile Time Null Checks

```
fun getCheckNulls(): Int{
    var a: String = "abc"
    a = null // compilation error

    var b: String? = "abc"
    b = null // ok

    b = getName()  // Call a function that returns a Nullable String
    b.length // Compilation Error, invoking method on potentially Null Object
    if (b != null){
        b.length // Can now execute method on the variable
    }

    // Safe Call Operator - Returns length or Null as an Int?
    var i = b?.length
```

Null chaining

```
val city = order?.customer?.address?.city

val city = order?.customer?.address?.city
           ?: throw IllegalArgumentException("Invalid Order")
```

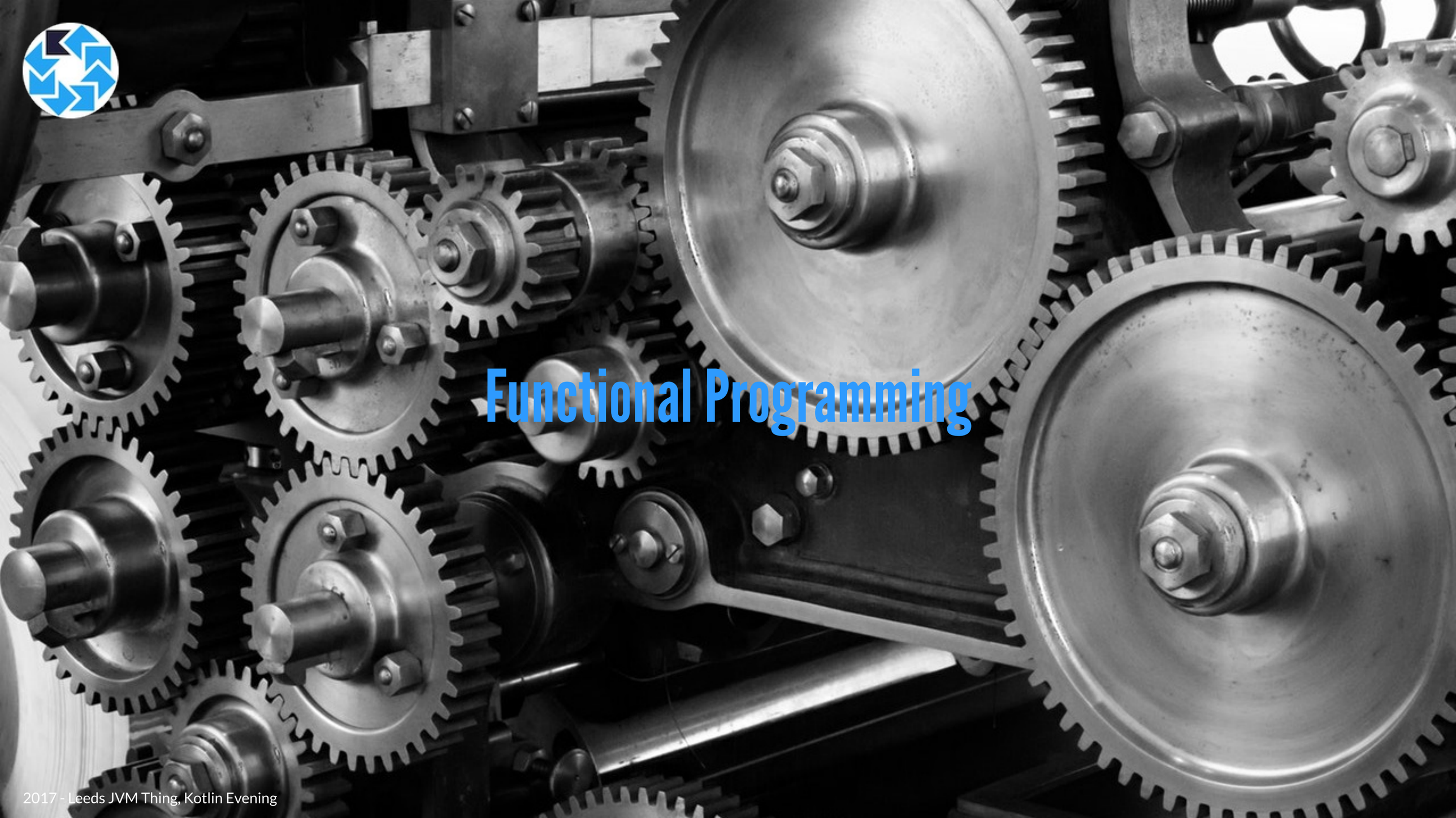
Smart Casting

- Type checks with the is or !is operator
- Compiler tracks is checks and performs automatic cast
- No need to create extra variables for cast results

Smart Casting Examples

```
fun demo(x: Any, y:Any?) {

    if (x is String) {
        print(x.length)
    }

    if (x !is String) return
    print(x.length) // x is automatically cast to String

    when (x) {
        is Int -> print(x + 1)
        is String -> print(x.length + 1)
        is IntArray -> print(x.sum())
    }

    val answer: String = y as String
```

2017 - Leeds JVM Thing, Kotlin Evening



Functional Programming

Higher Order Functions

- Kotlin supports functional programming
- Kotlin functions can:
  - Accept functions as parameters
  - Return functions as results
- Functions can also be declared as variables

Higher Order Functions - Example

```
fun getTaxCalculation(state:String) : (BigDecimal) -> BigDecimal {
    ...
}

fun BigDecimal.calculateTax(calculator: (BigDecimal) -> BigDecimal) = calculator(this)

val calculator = getTaxCalculation("CA")
val orderValue = BigDecimal(1789.25)
val taxValue = orderValue.calculateTax(calculator)
```

Lambda Functions

- Kotlin supports lambda functions
- Cleaner syntax than Java
- No need for functional interfaces
- Any function passed as a parameter can be replaced by a lambda

Lamda Functions - Example

```
people.filter({person -> person.name.startsWith("S")})
people.filter{person -> person.name.startsWith("S")}
people.filter{it.name.startsWith("S")}
```

Collections

- Similar to Java 8 Streams
- Cleaner syntax than Java
- Fluent interface for
  - Filtering
  - Sorting
  - Mapping
  - Grouping/Partitioning
- Supports potentially infinite sequences

Collections Example

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

val people = peopleRepository.fetchAll()
val longestName = people.filter{ it.age < 20 }
        .map{ it.name }
        .maxBy{ it.length }
```

Functional Support

- In terms of inbuilt functional support
  - Java < Kotlin < Scala
- Additional functional support
  - funKTionale library
  - Future versions of Kotlin

2017 - Leeds JVM Thing, Kotlin Evening

Summary

- Kotlin addresses many issues with Java
- It is supported by JetBrains & Google
- It combines OO & Functional programming
  - It's Pragmatic & Productive
- Gentle learning curve
- Kotlin is ready for prime time

Further Reading

- Kotlin Home Page: http://kotlinlang.org/
- Slack Channel : kotlinlang
- Book: Kotlin in Action
  - Dmitry Jemerov and Svetlana Isakova
- Udemy Course : https://www.udemy.com/kotlin-course/
- Twitter : @Kotlin
- Kotlin Yorkshire : @KotlinYorkshire
  - https://www.meetup.com/Kotlin-Yorkshire-Meetup-Group/