# From Groovy to Java 8

Dan Woods

# Who am I?

NETFLIX

Gmail by Google danielpwoods@gmail.com

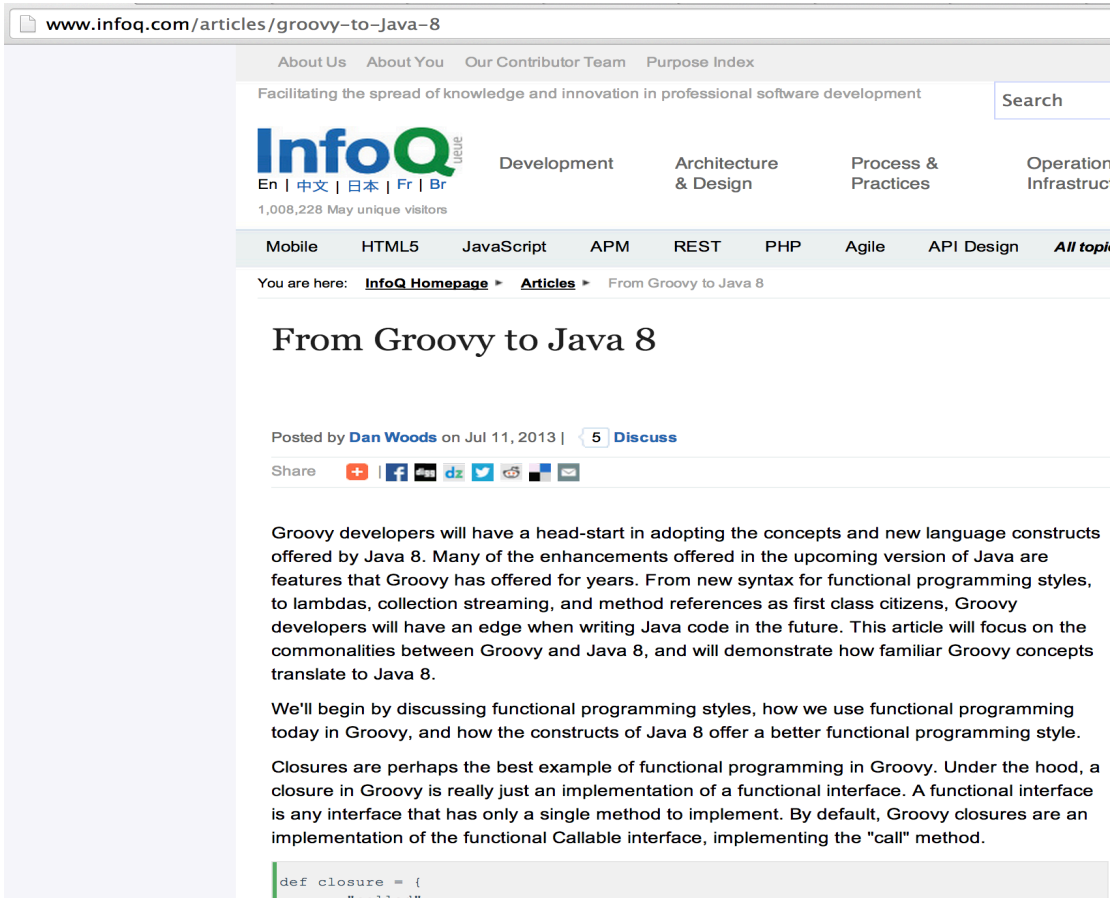
InfoQ #editor

/danveloper

@danveloper

# Where did this start?

http://www.infoq.com/articles/groovy-to-java-8





# What is the purpose of this talk?

- This is *NOT* to show you how to switch from Groovy to Java 8
- Talk is designed to introduce Groovy developers to Java 8
- So much new stuff in Java 8… Understand it on familiar terms
- Maybe you will learn cool new Groovy stuff!



Enter Functional Programming…



**Closures in Groovy…**

- Enable you to pass functionality as an argument to a method
- Allow you to treat code as data
- Simplify the syntax in your application!
- Can also “close over” state



**More Groovy Closures…**

- Can take a delegate of any object type
- Support currying



**Lambda Expressions…**

- Enable you to pass functionality as an argument to a method
- Allow you to treat code as data
- Beyond that…
- Simply the syntax in your application!



**Lambda Expressions…**

- Shares the single-line-no-return-statement-required methodology with Groovy
- Can be automatically coerced to a SAM interface
- More work needs to be done to allow them to retain state
- Scope cannot be configured



## Groovy Method References

- Allow methods to be coerced to Closure types
- Can be coerced to different types afterwards
- Method Closures can retain state from parent object, even in their new form



## Java Method *Handles*

- Coerces method to a lambda expression
- Can be further coerced to other functional types
- *Can ALSO* retain state from an instance method reference



## Functional Types in Groovy

- Abstract classes or functional interfaces
- Required that only a single method is able to be implemented
- Closures can be cast to this type, or inferred when being passed to a method call



**Functional Types in Java**

- Must be derived from a functional interface, with a single method
- A bunch of these exist for different reasons, right out of the box
- Type inference is automatic – no need for type casting!



Interface Defaults

**this. changes. EVERYTHING!**



## Interface Defaults

- Designed to decorate functionality to existing interfaces
- We can hack it to be much cooler
- Lambda expressions can access interface defaults
- Interface default methods can access “this”

# Hacking Java 8 for the Better

## Replicating Groovy Functionality



## Tail Call Optimization

- Groovy has tricks for emulating “tail call optimization”
- Employs the “trampoline” method to keep the stack from growing
- Very easy to do in Groovy
- We can do some trickery with Interface Defaults to get similar behavior from a lambda in Java 8 >:-]

# Replicating Groovy Functionality

**Memoization**

- Groovy Closures are given the ability to be very easily memoized
- Whereby, values will first attempt to be retrieved from cache, if not available, will be calculated
- Performance: don't do more work than necessary
- We can get this same behavior from lambdas in Java 8 with more trickery on Interface Defaults

*Aside:*

# Interface Defaults & Traits


# Collections



## Iterating Lists

- Moving the iterator from our code to the Collection API
- Groovy has had Collection#each since the beginning
- Java 8 has Groovy-style internal iterators, almost exactly



## Modifying Lists

- Groovy offers the Collection#collect method, which takes a Closure for each element
- Java 8's Stream API can be leveraged for *mapping* new values and *collecting* them into a new list



## Filtering Lists

- Groovy has the concept of *find* and *findAll* on a collection, which takes a closure that is coerced to a predicate
- Java 8's Stream API can be leveraged to *filter* a list, then *collect* the results



## Sorting Lists

- Groovy Collections have internal sorting methods, which can optionally take a closure that is coerced to a *Comparator*
- Java 8's Stream API offers a *sorted* method, which accepts a lambda expression that is also coerced to a *Comparator*


# Parallel Processing



## Parallel Streams

- In Groovy, we've been able to make use of the GPars library where we can parallelize using a special set of APIs
- Java 8 has a parallel stream API that will parallelize any of its regular stream functions



Differences: GPars & Java 8 Parallel Streams

- GPars can use a customizable thread pool for the processing, can take a fixed number or an executor
- Java 8 parallel streaming is driven by the ForkJoin pool, which pools based on the number of cores available

# Future for Groovy and Java 8 Interop



## Future of Groovy & Java 8 Interop

- There are two really big questions to ask about this:
  - Is Groovy going to support the Lambda expression *syntax?*
  - Is Groovy going to actually support Lambda expressions?

- Definitely will need to bring in support for Interface Defaults



Future of Groovy & Java 8 Interop

- Where are we at with Groovy?
- Is the value proposition still seamless interop with Java?
  - If so, this has already been broken in Java 7
- Where do we go from here?

# Recruiting


Come work with me on awesome DevOps stuff.

http://jobs.netflix.com