Learning 5 JVM Languages in the next 5 Years

Daniel Hinojosa

Hi, I'm polyglot Danno

So there is my commercial

If you limit your knowledge on one language

- You are purposefully rejecting great ideas of other languages.
- You are less likely to understand scripts is OSes and other language
- Less likely to fully understand operating system
- You are more likely to refuse to use great tools written in other languages
- Of course you are limiting your potential both financially and professionally

Alert: I was like Skeevy Dan

Pragmatic Programmer

The Pragmatic Programmer

from journeyman to master

Andrew Hunt
David Thomas

Foreword by Ward Cunningham

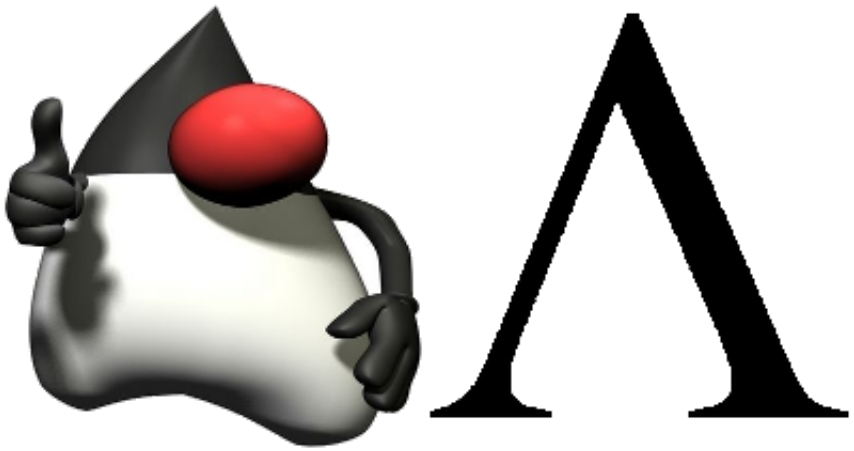
Learning 5 JVM Languages in the Next 5 Years.

Java

We know:

- It works very well
- It is a well cultivated language
- It brings us the money

Java 8 and Lambdas



Lambdas in version 8:

- Change the nature of the language and how you work with the language,
- Brings it up to par with what Groovy, JRuby, Clojure, and Scala have had for years.
- Learning some of the block or lambda constructs in other 4 languages you might find Java 8 Lambdas a bit verbose.

Who's to blame for such function verbosity?

- But Don't blame Java engineers for this, in fact congratulate them!
- They turned a rusty bucket of bolts and retrofitted it with some advanced features
- They did it *without* breaking the core of the language and how we understand it.

Custom Predicate

```
public interface MyPredicate<T> {
   public boolean apply(T t);
}
```

Using the `MyPredicate` interface

```
public static <T> List<T> myFilter (List<T> list, MyPredicate<T> predicate) {
    Iterator<T> it = list.iterator();
    List<T> result = new ArrayList<T>();
    while (it.hasNext()) {
        T next = it.next();
        if (predicate.apply(next)) {
           result.add(next);
        }
    }
    return result;
}
```

Calling `myFilter` with an implementation!

```
myFilter(Arrays.asList(1,2,3,4), new MyPredicate<Integer>() {
   public boolean apply(Integer i) {
      return i % 2 != 0;
   }
})
```

About Java 8 Lambdas

Functional Interface Definition

> *A functional interface is any interface that contains only one abstract method. (A functional interface may contain one or more default methods or static methods.) Because a functional interface contains only one abstract method, you can omit the name of that method when you implement it.*

(equals is an explicit declaration of a concrete method inherited from Object that, without this declaration, would otherwise be implicitly declared.)

Does this contain one abstract method?

```
public interface MyPredicate<T> {
   public boolean apply(T t);
}
```

Does this contain one abstract method?

```
public interface MyPredicate<T> {
   public boolean apply(T t); // Of course!
}
```

Therefore we can "lambdaize" `MyPredicate`

```
myFilter(Arrays.asList(1,2,3,4), i -> i % 2 != 0);
```

But Predicates and Filtering is already built in with a catch

```
Arrays.asList(1,2,3,4).stream().filter(i -> i % 2 != 0);
```

java.util.stream.ReferencePipeline$2@4617c264

Streams are continuous, therefore we need to recollect!

Collectors are terminal objects that convert a stream to something that you need.

Unfortunately…

All Methods	Static Methods	Instance Methods	Abstract Methods	Default Methods

Modifier and Type	Method and Description
`BiConsumer<A,T>`	**`accumulator()`** A function that folds a value into a mutable result container.
`Set<Collector.Characteristics>`	**`characteristics()`** Returns a `Set` of `Collector.Characteristics` indicating the characteristics of this Collector.
`BinaryOperator<A>`	**`combiner()`** A function that accepts two partial results and merges them.
`Function<A,R>`	**`finisher()`** Perform the final transformation from the intermediate accumulation type `A` to the final result type `R`.
`Supplier<A>`	**`supplier()`** A function that creates and returns a new mutable result container.

Thankfully there are prefab collectors

Both of the following will return: `[1,3]` with different collections

To a List

```
Arrays.asList(1,2,3,4).stream().filter(i -> i % 2 != 0)
      .collect(Collectors.toList());
```

To a Custom Collection

```
Arrays.asList(1,2,3,4).stream().filter(i -> i % 2 != 0)
      .collect(Collectors.toCollection(TreeSet::new))
```

Groovy

Lambdas in Groovy

```
Closure c = {x -> x % 2 != 0}
```

Closures Applied

```
Closure c = {x -> x % 2 != 0}

[1,2,3,4].grep(c) //[1,3]
```

Note: There are no collectors!

Inline Closures

```
[1,2,3,4].grep{x -> x % 2 != 0} // [1,3]
```

`it` is good

```
[1,2,3,4].grep{it % 2 != 0}
```

Collecting or Mapping in Groovy

```
[1,2,3,4].collect{it * 3}
```

Renders:

```
[3,6,9,12]
```

Collections and Mutability in Groovy

Given the following

```
import groovy.transform.*

@Canonical
class Employee {
 def firstName
 def lastName
 def salary
}
def hw = new Employee("Handome", "Wonderful", 40000)
def employees = [hw,
                 new Employee("Sarah", "Stylish", 50000),
                 new Employee("Sylvia", "Sleek", 80000)]

println(employees.collect{it.salary += 10000}) //[50000, 60000, 90000]
println(hw) //Employee(Handome, Wonderful, 50000)
```

The Spread Operator

```
import groovy.transform.*

@Canonical
class Employee {
 def firstName
 def lastName
 def salary
}
def hw = new Employee("Handome", "Wonderful", 40000)
def employees = [hw,
                 new Employee("Sarah", "Stylish", 50000),
                 new Employee("Sylvia", "Sleek", 80000)]

println(employees*.salary += 10000) //[50000, 60000, 90000]
println(hw) //Employee(Handome, Wonderful, 50000)
```

JRuby

Lambdas in JRuby

JRuby has 3 different forms of functions

Λ

Blocks in JRuby

```
array = [1, 2, 3, 4]

array.select do |x| x % 2 != 0
```

or

```
array = [1, 2, 3, 4]

array.select {|x| x % 2 != 0}
```

How does a JRuby/Ruby block work?

```
def call_block
  yield('hello', 99)
end
```

Then call either:

```
call_block {|str, num| puts str + ' ' + num.to_s}
```

or:

```
call_block do |str, num| puts str + ' ' + num.to_s end
```

Which both renders…

```
hello 99
```

Source: http://rubylearning.com/satishtalim/ruby_blocks.html

Multiline JRuby Blocks

```
def call_block
  yield('hello', 99)
end
```

Then call either:

```
call_block {|str, num|
   puts str + ' ' + num.to_s
}
```

or:

```
call_block do |str, num|
   puts str + ' ' + num.to_s
end
```

Which both renders…

```
hello 99
```

JRuby Procs

Unfortunately this is not possible …

```
def call_block
  yield('hello', 99)
end

myBlock = {|str, num| puts str + ' ' + num.to_s}
call_block myBlock
call_block myBlock
```

JRuby Procs

A block is a Ruby Proc, so we can just declare it as such.

```
def call_block(&block)
   block.call('hello', 99)
end

call_block {|str, num| puts str + ' ' + num.to_s}
call_block do |str, num| puts str + ' ' + num.to_s end
```

Coming back to our problem

```
def call_block
  yield('hello', 99)
end

myBlock = {|str, num| puts str + ' ' + num.to_s}
call_block myBlock
call_block myBlock
```

Our problem can be fixed by explicitly declaring using a Proc

```
def call_block(block)
   block.call('hello', 99)
end

my_block = Proc.new{|str, num| puts str + ' ' + num.to_s}

call_block my_block
call_block my_block
```

Note: `block` does not contain `&`

JRuby Lambdas

Ruby and JRuby have another construct, lambdas with different behaviors. When the lambda is returned, the rest of the block continues

```
def lambda_method
   internal_lambda = lambda {return 4}
   puts "starting"
   puts internal_lambda.call
   puts "ending"
end

puts lambda_method
```

```
starting
4
ending
<blank line>
```

What a standard Proc looks like when returned

```
def proc_method
   internal_proc = Proc.new {return 4}
   puts "starting"
   puts internal_proc.call
   puts "ending"
end

puts proc_method
```

```
starting
4
```

Lambdas in Ruby/JRuby are strict with the number of arguments

```
def read_block(block)
   block.call(4,3)
end
```

```
puts read_block Proc.new{|x, y, z| "x is #{x}, y is #{y}, and z is #{z}"}
x is 4, y is 3, and z is
```

```
puts read_block lambda{|x,y,z| "x is #{x}, y is #{y}, and z is #{z}"}
wrong number of arguments (2 for 3)
```

Note: Ruby/JRuby functional lambda verdict? A wee bit shaky

Clojure

Standard Function in Clojure

```
(defn average
  [numbers]
    (/ (apply + numbers) (count numbers)))
```

```
(average [1.0 3.0 10.4 11.5])
```

Calling Clojure with higher order functions

```
(defn op [f a b] (f a b))
```

```
(op + 4 2)
```

Anonymous Functions in Clojure

```
(map (fn [i] (* i 2)) [1 2 3 4 5])
```

```
[2 4 5 8 10]
```

Anonymous Functions in Clojure

```
(map #(* %1 2) [1 2 3 4 5])
```

```
[2 4 6 8 10]
```

More Anonymous Function Variants in Clojure

```
(map #(* % 2) [1 2 3 4 5])
```

Scala

Functions in Scala

```
val f = (x:Int) => x % 2 != 0

List(1,2,3,4).filter(f)
```

Functions in Scala with an alternate List style

```
val f = (x:Int) => x % 2 != 0

(1 :: 2 :: 3 :: 4 :: Nil).filter(f)
```

Inline Functions in Scala

```
List(1,2,3,4).filter((x:Int) => x % 2 != 0)
```

or with type inference

```
List(1,2,3,4).filter(x => x % 2 != 0)
```

Inline Function Variant in Scala

```
List(1,2,3,4).filter(_ % 2 != 0)
```

Collections

Java Lists

```
List<Employee> employees = new ArrayList<Employee>();
employees.add(...);
employees.add(...);
```

or

```
Arrays.asList(new Employee(...),new Employee(...), new Employee(...))
```

Java Sets

```
Set<Employee> employeeSet = new HashSet<Employee>();
employeeSet.add(...);
employeeSet.add(...);
employeeSet.add(...);
```

Java Maps

```
Map<Int, String> maps = new HashMap<Int, String>();
maps.put(1, "One");
maps.put(2, "Two");
maps.put(3, "Three");
```

Groovy Lists

```
def list = [1,2,3,4]
```

Groovy Sets

```
def set = [1,2,3,4] as HashSet
```

Groovy Maps

```
def map = [1 : "One", 2 : "Two", 3 : "Three"]
```

JRuby/Ruby Arrays

```
a = [1,2,3,4]
```

JRuby/Ruby Sets

```
require 'set'
s = [1,2,3,4].to_set
s1 = Set.new [1,2,3,4]
```

JRuby/Ruby Hashes

```
h = {1 => 'One', 2 => 'Two', 3 => 'Three'}
```

Scala Lists

```
val list = List(1,2,3,4)
val list2 = 1 :: 2 :: 3 :: 4 :: Nil
```

Scala Sets

```
val set = Set(1,2,3,4)
```

Scala Maps

```
val Map = Map(1 -> "One", 2 -> "Two", 3 -> "Three")
```

Clojure Lists

```
(def a-list (list 1 2 3 4))
(def states '("Ohio" "Michigan" "Illinois" "Indiana"))
```

Clojurists don’t use lists for collections

```
(def states ["Ohio" "Michigan" "Illinois" "Indiana"])
```

Note: Lists are used throughout the language as core functionality

Clojure Sets

```
(def s #{1 2 3 4})
```

Clojure Maps

```
(def national-league {:reds "Cincinnati Reds" :dodgers "Los Angeles Dodgers"
                      :braves "Atlanta Braves" :astros "Houston Astros"
                      :expos nil})
```

Classes, Methods, Member Variables

Java 8 Classes

Lot of work!

```
public class Employee {
   private String firstName;
   private String lastName;
   public Employee(String firstName, String lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }
   public void setFirstName(String firstName) {...}
   public String getFirstName() {...}
   public void setLastName(String lastName) {...}
   public String getLastName() {...}
   public int hashCode() {...}
   public String toString {...}
   public boolean equals(Object o) {...}
}
```

Groovy Classes

```
@Canonical
class Employee {
   def firstName
   def lastName
}
```

Ruby/JRuby Classes

```
class Employee
   def initialize(first_name, last_name)
      @first_name = first_name
      @last_name = last_name
   end

   def first_name
      @first_name
   end

   def last_name
      @last_name
   end

   def to_s
      "Employee(#{@first_name} #{@last_name})"
   end

   def ==(other)
      other.first_name == @first_name
      other.last_name == @last_name
   end

   def hash
      @first_name.hash ^ @last_name.hash
   end
end
```

Note: Initialize sets the member variable, there is no declaration

Ruby/JRuby Classes

```
class Employee
   attr_reader :first_name, :last_name

   def initialize(first_name, last_name)
      @first_name = first_name
      @last_name = last_name
   end

   def to_s
      "Employee(#{@first_name} #{@last_name})"
   end

   def ==(other)
      other.first_name == @first_name
      other.last_name == @last_name
   end

   def hash
      @first_name.hash ^ @last_name.hash
   end
end
```

Note: `attr_reader` does not create member variables!

Setting member variables in Ruby

```
class Employee
   attr_reader :first_name, :last_name
   attr_writer :first_name, :last_name

   def initialize(first_name, last_name)
      @first_name = first_name
      @last_name = last_name
   end#

   def to_s
      "Employee(#{@first_name} #{@last_name})"
   end

   def ==(other)
      other.first_name == @first_name
      other.last_name == @last_name
   end

   def hash
      @first_name.hash ^ @last_name.hash
   end
end
```

Note: `attr_writer` does not create member variables!

Accessor member variables in Ruby

```
class Employee
   attr_accessor :first_name, :last_name

   def initialize(first_name, last_name)
      @first_name = first_name
      @last_name = last_name
   end#

   def to_s
      "Employee(#{@first_name} #{@last_name})"
   end

   def ==(other)
      other.first_name == @first_name
      other.last_name == @last_name
   end

   def hash
      @first_name.hash ^ @last_name.hash
   end
end
```

The Ugly Mutable Scala Class

```
class Employee {
   var fn:String = _
   var ln:String = _
   def firstName = fn
   def lastName = ln
   def firstName_=(x:String):Unit = {fn = x}
   def lastName_=(x:String):Unit = {ln = x}
   override def equals(x:Any):Boolean = {
      x match {
        case x:Employee => (x.fn == fn && x.ln == ln)
        case _ => false
      }
   }
   override def hashCode:Int = (fn.hashCode ^ ln.hashCode) + 49
   override def toString:String = s"Employee($fn, $ln)"
}
```

Introducing Immutability in the Scala Class

```
class Employee (val firstName:String, val lastName:String){
   override def equals(x:Any):Boolean = {
      x match {
        case x:Employee => (x.fn == fn && x.ln == ln)
        case _ => false
      }
   }
   override def hashCode:Int = (fn.hashCode ^ ln.hashCode) + 49
   override def toString:String = s"Employee($fn, $ln)"
}
```

Scala Case Classes

```
case class Employee(firstName:String, lastName:String)
```

Note: Awesome!

Clojure Classes?

- There are five different way to create "types" in Clojure
  - `deftype`
  - `reify`
  - `gen-class`
  - `proxy`
  - `defrecord`
- Most are used to interop with Java
- The most important point about Clojure is, it's not really an OOP
- Most Clojurists keep it simple and use maps to encapsulate "models"

Maps as Models

```
(def employee_of_the_month {:first-name "Milton" :last-name "Waddams"})

(println (:first-name employee_of_the_month)) => "Milton"

(println (employee_of_the_month :first-name)) => "Milton" ; Whoa! A Map is a function!
```

`defrecord` in Clojure

- `defrecord`

- Compiles to a class of the same name, and inside same package
- Contains the same functionality as a map

```
(defrecord Employee [first-name last-name])

(def roger (Employee. "Roger" "Moore"))

(:last-name roger) ; "Moore"
```

`deftype` in Clojure

- `deftype`
  - Compiles to a class of the same name, and inside same package, like `defrecord`
  - Provides no functionality not specified by the user, other than a constructor
  - Supports immutable fields

```
(deftype Employee [first-name last-name])

(def employee (Employee. "Maya" "Angelou"))

(println (.first-name employee))
```

Apps & Libraries

Java Products

- Thousands upon Thousands of Applications
- Android
- Spring Framework and Libraries (Spring core, Spring Security, Spring Boot, Spring Batch…) (http://w
- JBoss Applications and Libraries (Hibernate ORM, Hibernate Search, Infinispan, …)
- Slew of Apache Products (Apache Lucene, Apache Solr, Apache Log4j, …)
- Google Libraries and Cloud Products (Guava)
- Enormous number of Java EE Products

Groovy Products

- Grails (grails.org)
- Gradle (gradle.org)
- Spock (https://github.com/spockframework/spock)

JRuby Products

- Buildr (http://buildr.apache.org/)
- Asciidoctor (http://www.asciidoctor.org)
- Ruby Developed Applications
  - Ruby on Rails (http://www.rubyonrails.org)
  - Rake (https://github.com/ruby/rake)
  - Cucumber (http://cukes.info)

Scala Products

- Play Framework (http://www.playframework.org)
- Akka (http://akka.io)
- Apache Spark (https://spark.apache.org/)
- LiftWeb (http://liftweb.net)

Clojure Products

- Datomic (http://www.datomic.com/)
- ClojureScript (https://github.com/clojure/clojurescript)

The Future of Development

> *In the future, a great programmer will be using something from all 5 of these languages*

REPL Driven Development

Following Language have REPL (Read Eval Print Loop)

- Groovy (groovy)
- JRuby (jirb)
- Clojure (clojure.main)

- Scala (scala)

Note: Groovy also has Groovy Console a GUI based evaluator

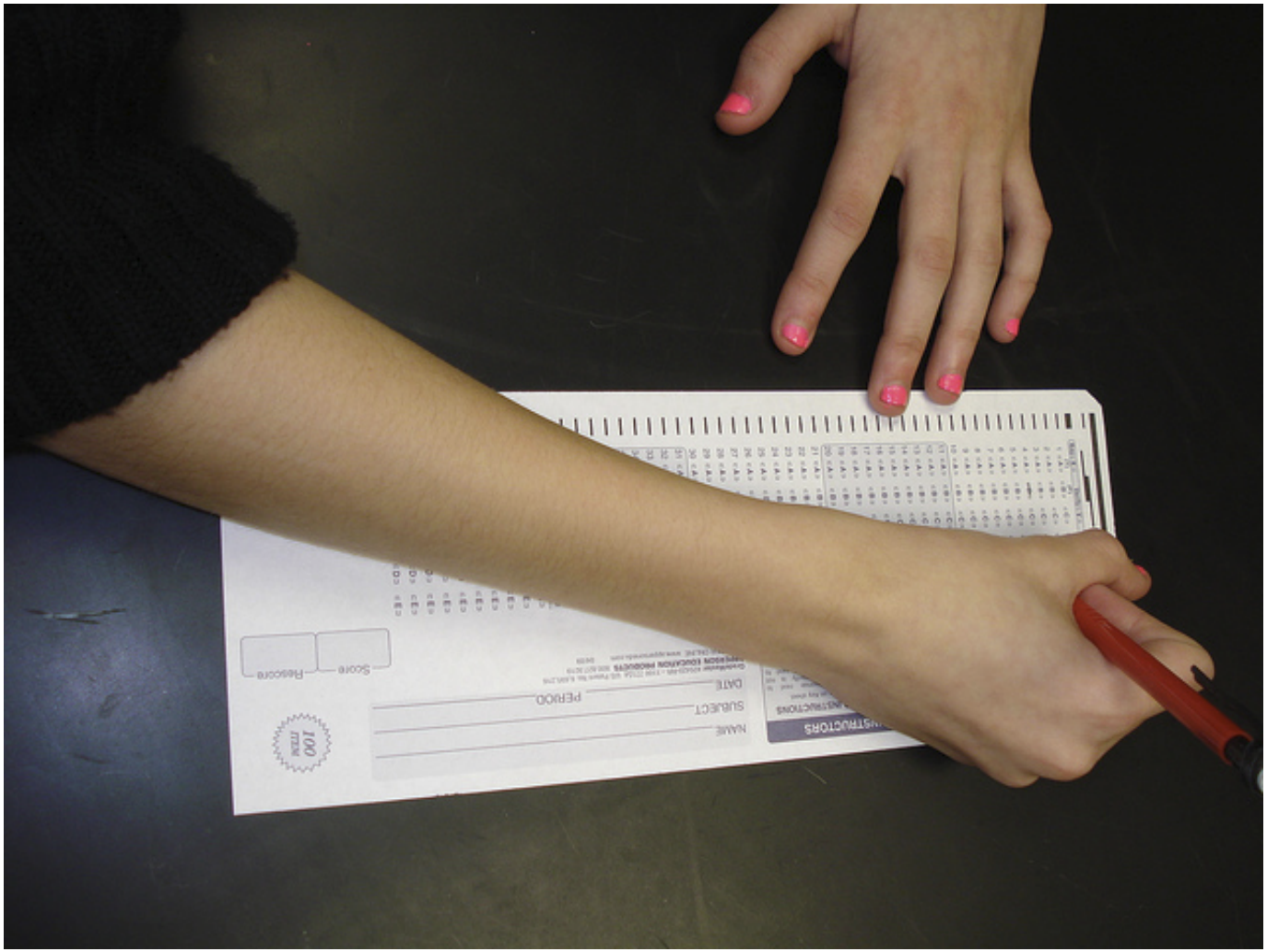
Learning

Learning the 5 different languages

- Lower learning curve languages
  - Groovy
  - Ruby
- Higher learning curve languages
  - Java 8
  - Clojure
  - Scala (highest)

How do I find time to learn all these languages?

Testing



Pomodoro Technique

Create Your Own Language Matrix

dhinojosa / **language-matrix**

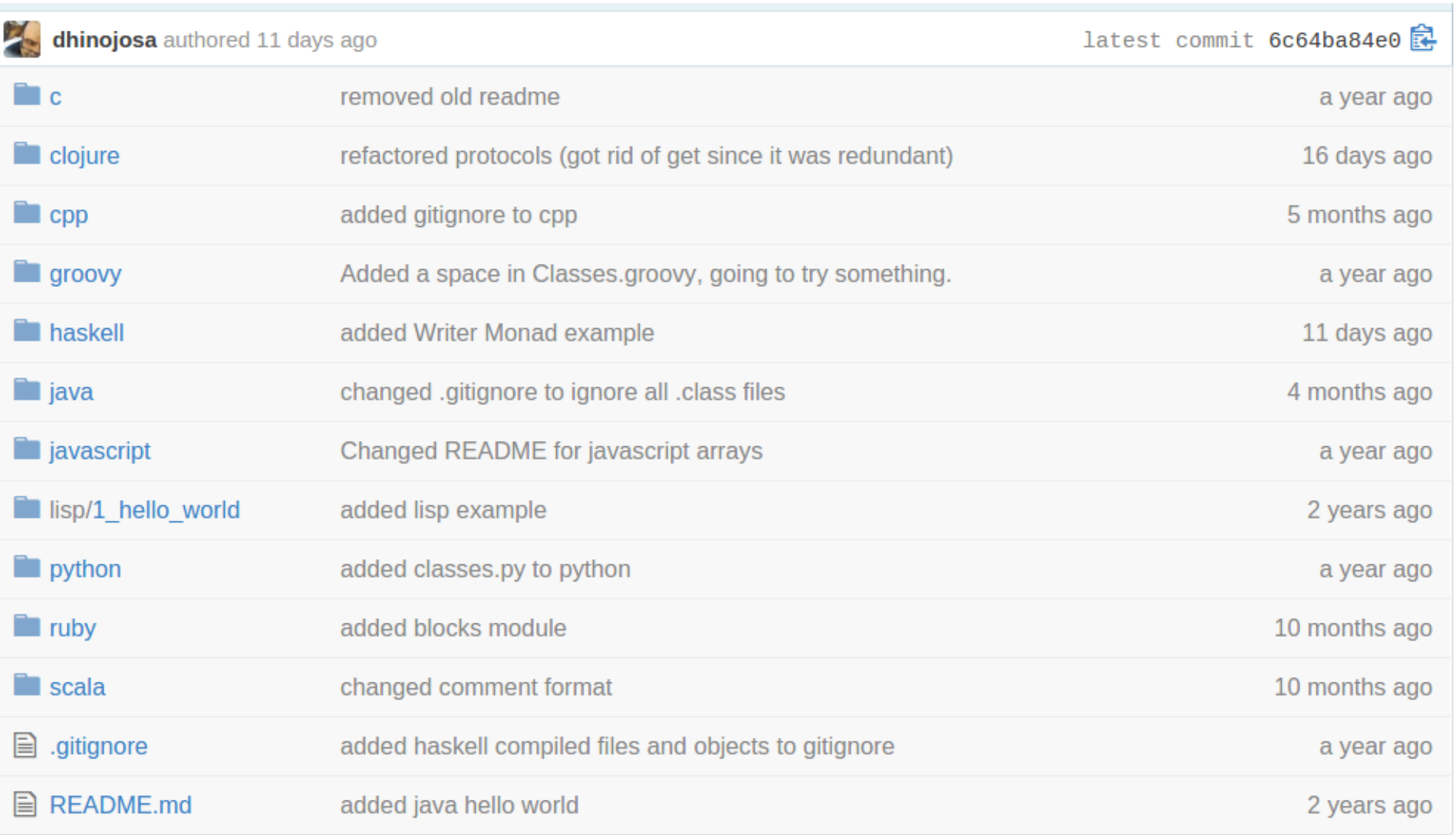
Unwatch 2

Code snippets to do useful things for 11 languages: C, C++, Groovy, Ruby, Java, Scala, Python, Haskell, Javascript, Lisp, Clojure — Edit

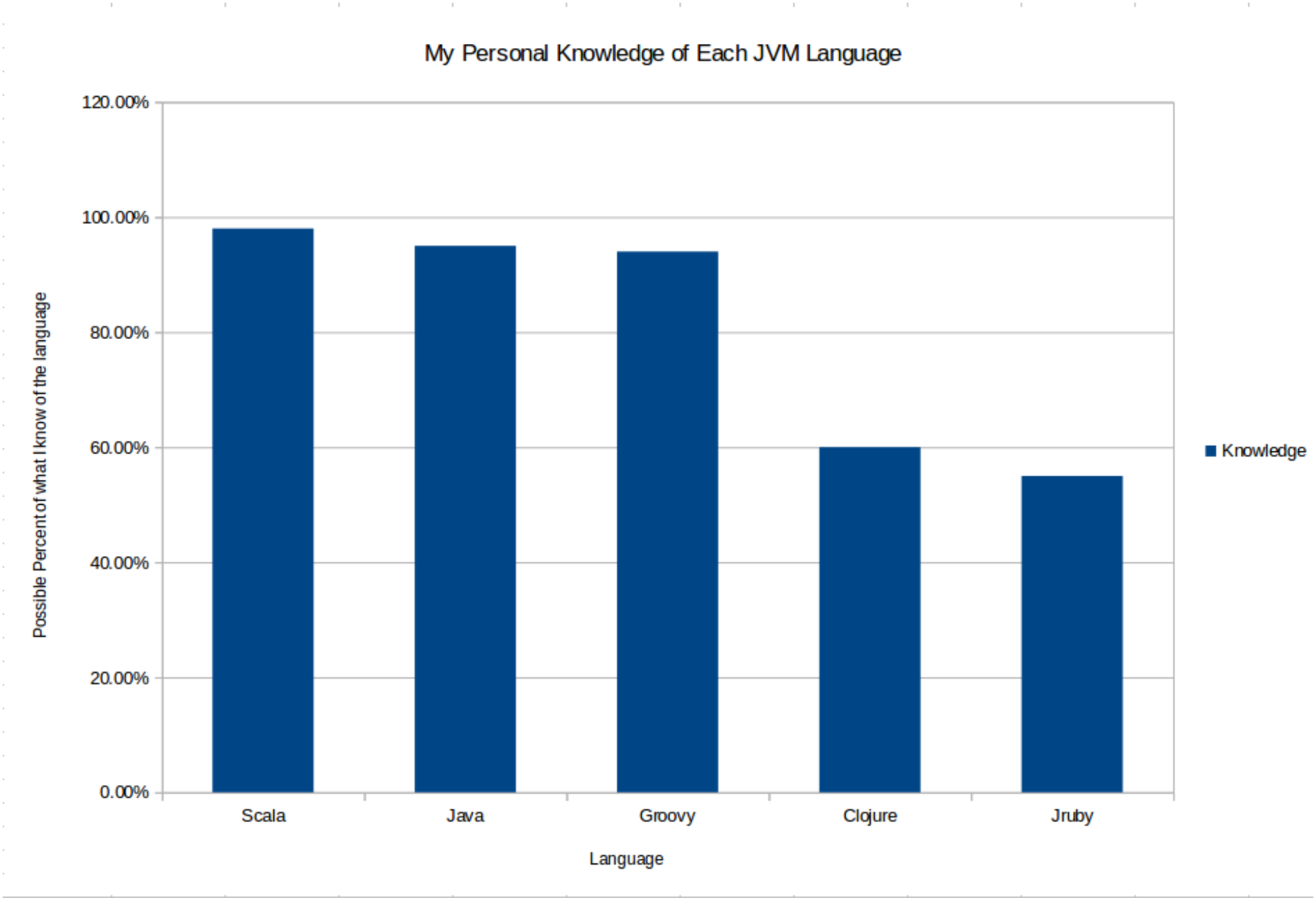
205 commits **1** branch **0** releases **1** contributor

branch: **master** language-matrix / +

added Writer Monad example



My Person Knowledge Graph


My Personal Knowledge of Each JVM Language
120.00%
100.00%
80.00%
60.00%
40.00%
20.00%
0.00%
Possible Percent of what I know of the language
Scala
Java
Groovy
Clojure
Jruby
Language
Knowledge

Culture of each language (Opinionated)

- Scala
  - Highly mathematical group
  - Fixated on functional correctness
  - Fixated on monadic structures
  - Community mixed bag
- Clojure
  - Highly functional
  - LISP oriented group
  - Fixated on homoiconicity - Representation of code as data!
  - Fixated on simplicity
  - Community very friendly

Culture of each language (Opinionated)

- JRuby / Ruby
  - Highly Creative Developers
  - Driven to create some of best webapps in history
  - Driven to create some great tools and ideas that are used throughout
  - Rails, Rake are dominant
  - Ruby Community (not JRuby)
    - Very friendly after 2005
    - Will sometimes press the bounds often calling things "dead" that are perfectly alive

Culture of each language (Opinionated)

- Java
  - Greatest Community Ever (please note this on reviews)
  - Very Imperative, almost C++ mental frame of mind
  - Changed the landscape, Spring in Particular
  - Reluctant to Change, especially when upgrading to later Java versions
  - Dominant Developer Culture World Wide!
  - You will always have a job with Java.
  - Always friendly

Groovy Culture (Opinionated)

- Underdog culture, Groovy doesn’t get enough credit
- Groovy is a very powerful, concise, expressive language
- Competes well with other languages, extremely clever
- Grails, Gradle, and Spock
- Netflix’s secret weapon

In the end I think you are going to love these two languages

But then again, it will be up to to decide which ones will be your choice?

“*But please, prove me wrong*

Questions?

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Twitter: http://twitter.com/dhinojosa
- Google Plus: http://gplus.to/dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext

Last updated 2015-03-11 10:10:19 EDT