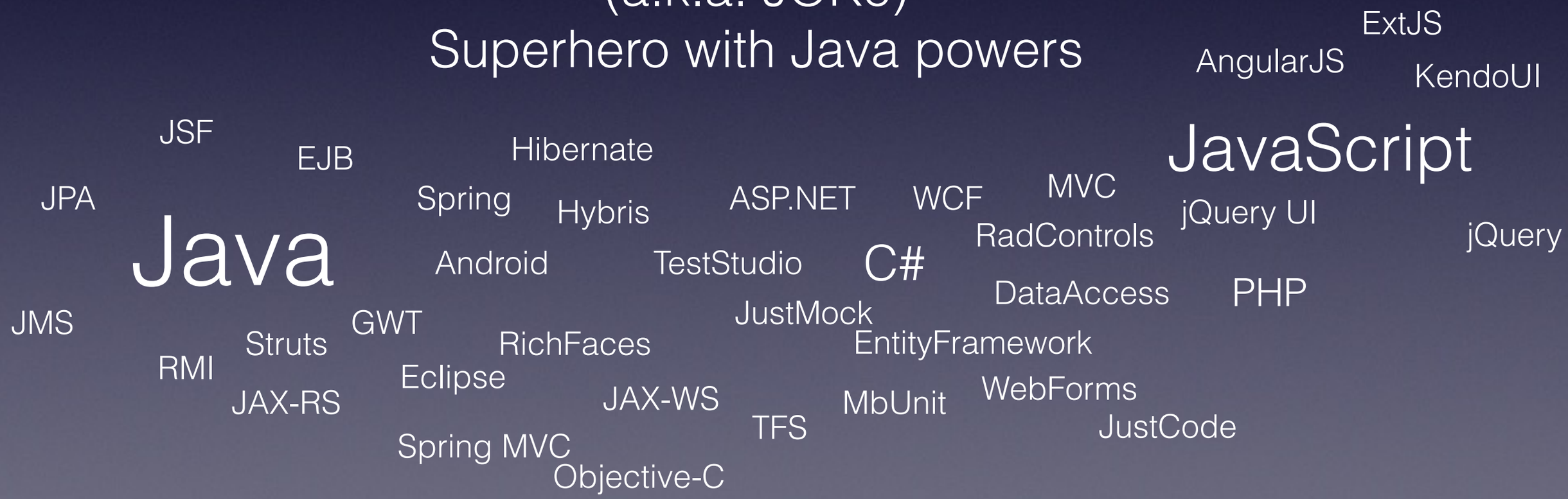
Pragmatic: helping you bypass unwanted jobs since 2013

Who am I ?

Nayden Gochev

(a.k.a. JOKe)

Superhero with Java powers



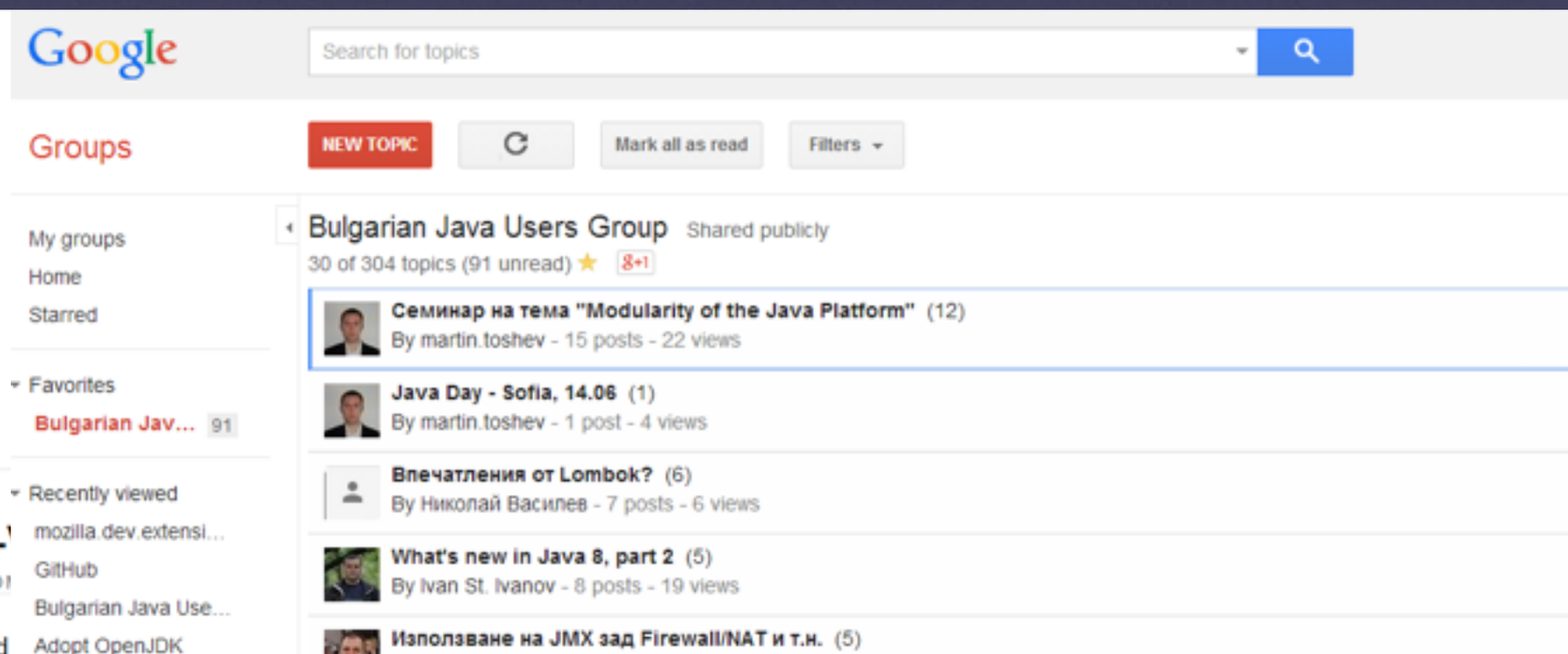
Java 8 hidden bits

Not your grandpa's Java

Special thanks to
Dmitry Alexandrov
Ivan St. Ivanov

But before that…
The Bulgarian Java User Group

- http://java-bg.org
- http://groups.google.com/group/bg-jug



So the plan.. the plan …was

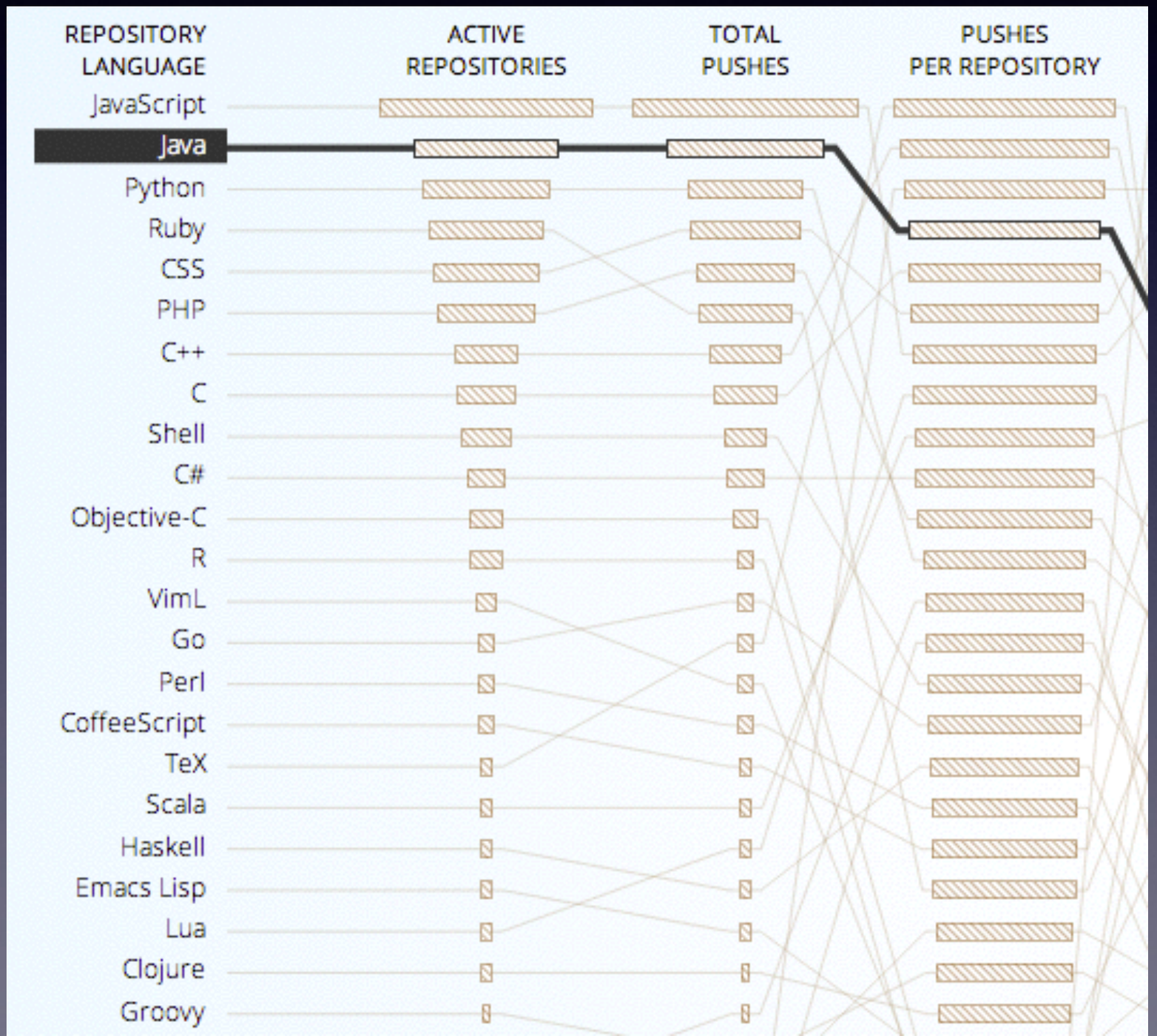
- Java 7 … yes 7
- Java 8 (you should know already about lambdas, stream API so on)
  - How this lambdas work ? anonymous class loader, invoke dynamic ?
  - Exact Numeric Operations
  - Type Annotations
  - Optional … or what the hell is this ?
  - “New” Date and Time .. another one !?
  - Nashorn

But it’s gone :(

The NEW plan is…

- Starting with Java 7 (1-3-4 slides)
- Java 8
  - Lambdas (what is this ? you saw yesterday right ? hmmm…)
  - Default methods on interfaces ? yes it is ugly :((cry :(
  - Static methods on interfaces (whaaat !?!?! Noooo)
  - Stream API (nice name isn't it ? ok LINQ is maybe nicer :)
  - Optional
  - Exact Numeric Operations
  - Type Annotations - we “should” have time for this
  - Nashorn - I really hope to have time for this … its awesome

But wait wait.. Java ? RIy? Why should we care?

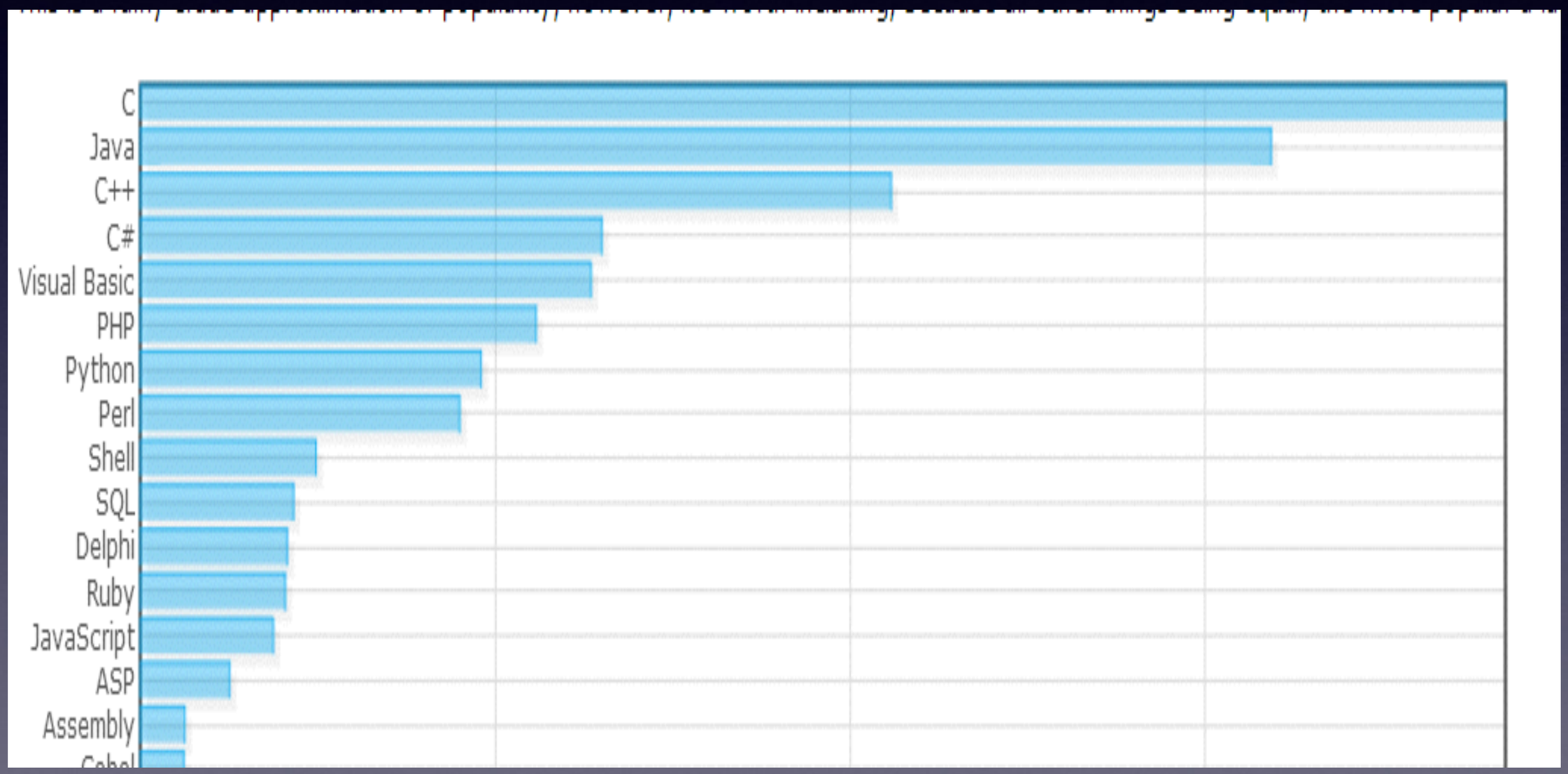


TIOBE Index for June 2014

Jun 2014	Jun 2013	Change	Language	Ratings	Change
1	1		C	16.191%	-1.62%
2	2		Java	16.113%	-0.54%
3	3		Objective-C	10.934%	+0.58%
4	4		C++	6.425%	-2.39%
5	6		C#	3.944%	-1.84%
6	7		(Visual) Basic	3.736%	-0.61%
7	5		PHP	2.848%	-3.14%

http://langpop.com/

Google Search Results



PYPL populartY index

- The PYPL PopularitY of Programming Language Index is created by analyzing how often language tutorials are searched

Position June 2014	Position June 2013	Delta in position	Programming language	Share in June 2014	Twelve mo nth trends
1	1		Java	26.9 %	-0.1 %
2	2		PHP	13.2 %	-1.4 %
3	4		C#	10.3 %	+0.2 %
4	3		Python	10.2 %	+0.4 %
5	5		C++	8.4 %	-0.9 %
6	6		C	7.9 %	-0.2 %

So .. yes… you should care !

But …Java is so old (~20 years in fact)

- “There are simply no new features in Java,”
- “Java looks like Y language X years ago”
- “C#, Ruby and Python has everything you can dream of. “

Reading is more important then writing.

“Code should be written for humans to read and only incidentally for computes to execute”

Java 7

- We all know about diamond operator right?
  - **List**<**Integer**> primes = **new ArrayList**<>();
- And Strings in switch
- And Fork Join Framework
- And Underscore in Numeric literals
  - int billion = **1_000_000_000**; // 10^9
- And maybe catching Multiple Exception Type in Single Catch Block
  - **catch**(**ClassNotFoundException**|**SQLException** ex)
- Binary Literals with prefix "0b“
  - int binary = **0**B0101_0000_1010_0010_1101_0000_1010_0010;
- G1 Garbage Collector
- And we all know about Automatic Resource Management

The biggest two additions to Java 7 are.. ?

1) MethodHandle

Method handles gives us unrestricted capabilities for calling non-public methods.

Compared to using the Reflection API, access checking is performed when the method handle is created as opposed to every time the method is called.

Reflection example

```
Class<?>[] argTypes =
                    new Class[] { int.class,
String.class };
Method meth = MethodAccessExampleWithArgs.class.
                   getDeclaredMethod(
                      "bar", argTypes);
meth.setAccessible(true);
meth.invoke(instance, 7, "Boba Fett");
```

MethodHandle example

```
MethodType desc = MethodType.methodType(
            void.class, int.class, String.class);
MethodHandle mh =   MethodHandles.lookup().findVirtual(
            MethodAccessExampleWithArgs.class, "bar", desc);
mh.invokeExact(mh0, 42, "R2D2");
```

and the second big thing in Java 7 is … ?

- 2) invokedynamic

The culmination of invokedynamic is, of course, the ability to make a dynamic call that the JVM not only recognizes, but also optimizes in the same way it optimizes plain old static-typed calls.

What we all (almost all?) know that before java 7 there were 4 byte codes for method invocations

- **invokevirtual** - Invokes a method on a class.
- **invokeinterface** - Invokes a method on an interface.
- **invokestatic** - Invokes a static method on a class.
- **invokespecial** - Everything else called this way can be constructors, superclass methods, or private methods.

how invokedynamic look

```
invokevirtual #4 //Method java/io/PrintStream.println:
(Ljava/lang/String;)V

invokedynamic #10 //NameAndTypelessThan:(Ljava/lang/
Object;Ljava/lang/Object;)
```

But wait. How does the JVM find the method if the receiver type isn't supplied?

When the JVM sees an invokedynamic bytecode, it uses the new linkage mechanism that I tried to explain yesterday. For more checkout the Brian Goetz JavaOne speak from 2013

Java 8

Lambdas

- Lambdas bring anonymous function types in Java (JSR 335):

```
(parameters) -> {body}
```

- Example:

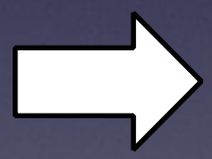
```
(x,y) -> x + y
```

Lambdas (2)

- Lambdas can be used in place of functional interfaces (interfaces with just one method such as **Runnable**)
- Example:

```
new Thread(new Runnable() {

@Override
public void run() {
    System.out.println("It runs!");
  }
}
).start();
```



```
new Thread(() ->
System.out.println("It runs!")
).start();
```

So what can be used as a lambda ?

- Every interface with only one method aka @FunctionalInterface
- Examples of such functional interfaces:

java.lang.Runnable -> run()
java.util.concurrent.Callable -> call()
java.security.PrivilegedAction -> run()
java.util.Comparator -> compare(T o1,T o2)
java.awt.event.ActionListener ->
actionPerformed (ActionEvent e)

Some new functional interfaces

- Additional functional interfaces are provided by the **java.util.function** package for use by lambdas such as:
  - **Predicate<T>** - one method with param of type T and boolean return type
  - **Consumer<T>** - one method with param T and no return type
  - **Function<T, R>** - one method with param T and return type R
  - **Supplier<T>** - one method with no params and return type T

Default methods

- Default aka extension methods provide a mechanism for extending an existing interface without breaking backward compatibility

```
public interface Iterable<T> {

 default void forEach(Consumer<? super T> action) {
       Objects.requireNonNull(action);
       for (T t : this) {
           action.accept(t);
       }
 }
}
```

Static methods on interfaces

- You can declare static methods on interfaces:

```
public interface ContentFormatter {
    public void format();
    static String convertRegex(String
        regex) {
            . . .
    }
}
```

- None of the implementing classes will have this method available to them
- The idea : less utility classes..

Notes about @FuncitonalInterfaces

```
@FunctionalInterface
interface Test2 {
  public default void doSomething() {}
}
```

This is NOT a functional interface and cannot be used as lambda

However this are valid functional interfaces

```
@FunctionalInterface
interface Test3 {
  public void doSomething();
  public default void doSomethingDefault() {}
}

@FunctionalInterface
interface Test4 {
  public void doSomething();
  public static void doSomethingDefault() {}
}
```

Last note about @FunctionalInterface(s)

This is valid

```
@FunctionalInterface
interface Test6<T> {
  public T doSomething();
}
```

But this is invalid:

```
@FunctionalInterface
interface Test6 {
  public <T> T doSomething();
}
```

2 notes about using Lambdas on an API

- Lets assume we had the following method pre Java 8

```
public static void doIt(Integer a){}
```

- Let assume now.. That we want this integer to be fetched lazy or via some method internally so we may add another method like this:

```
public static void doIt(Callable<Integer> a){ }
```

So far so good.

However pre Java 8 we had a chance to do this for the next API version:

```
public static void doIt(Integer a){
}
public static void doIt(String a){
}
```

But now .. we cant add this:

```
public static void doIt (Callable<Integer> a){
}
public static void doIt (Callable<String> a){
}
```

Because the generics are removed and this two methods became the same so we got duplicated method. So this is something we need to think about when designing API.

Second note: Lets say we had only one method with one generic type but we want to use another interface

- So we had:

```
public static void doIt(Callable<Integer> a){
}
```

- and we may add

```
public static void doIt(Supplier<Integer> a){
}
```

- And that’s look fine… to us.

However when we try to call this method(s)

```
doIt(() -> 5);
```

Now we (in fact all clients that used our API) get ambigous method.

And the only workaround for them is to change their code to something like :

```
doIt((Callable<Integer>)() -> 5);
```

Which is… if first ugly and second it require manual change(s) in the client code.

Stream API

- Databases and other programming languages allow us to specify aggregate operations explicitly
- The stream API provides this mechanism in the Java platform
- The notion of streams is derived from functional programming languages

Stream API (2)

- The stream API makes use of lambdas and default methods
- Streams can be applied on collections, arrays, IO streams and generator functions
- Streams can be finite or infinite
- Streams can apply intermediate functions on the data that produce another stream (e.g. map, reduce)

Stream API usage

java.util.stream.Stream<T> **collection.stream();**

java.util.stream.Stream<T> **collection.parallelStream();**

Stream API useful methods

- filter(Predicate), map(Function), reduce(BinaryOperator), collect(Collector)
- min(Comparator), max(Comparator), count()
- forEach(Consumer)
- findAny(), findFirst()
- average(), sum()

Prior to Java 8, print even numbers

```
List<Integer> ints = Arrays.asList(1, 2, 3, 4,
    5, 6);
for (Integer anInt : ints) {
    if (anInt % 2 == 0) {
        System.out.println(anInt);
    }
}
```

After java 8

```
List<Integer> ints = Arrays.asList(1, 2, 3, 4,
    5, 6);
ints.stream()
    .filter(i -> i % 2 == 0)
    .foreach(i -> System.out.println(i));
```

Optional

Optional

The main point behind Optional is to wrap an Object and to provide convenience API to handle nullability in a fluent manner.

```
Optional<String> stringOrNot =
Optional.of("123");

//This String reference will never be null

String alwaysAString = stringOrNot.orElse("");
```

Optional in a nice example by Venkat Subramanian

- Task : Double the first even number greater than 3
- Having as example the following input

```
List<Integer> values = Arrays.asList(1,2,3,4,5,6,7,8,9,10);
```

Pre Java 8 way

```
int result = 0;
for(int e : values){
    if(e > 3 && e% 2 == 0) {
                result = e * 2;
                break;
                }
    }
    System.out.println(result);
```

Is it correct ? Is it ok ? .. Lets start eclipse and check

The new lambda way

```
System.out.println(
      values.stream()
        .filter(value -> value >3)
.filter(value -> value % 2 == 0)
.map(value -> value * 2)
.findFirst() );
```

Ones you write this 10 times its not that hard?

So in summary Optional is heavily used in Stream API

```
// This Integer reference will be wrapped again
Optional<Integer> integerOrNot =
    stringOrNot.map(Integer::parseInt);
```

```
// This int reference will never be null
int alwaysAnInt = stringOrNot
    .map(s -> Integer.parseInt(s))
    .orElse(0);
```

Method references

- Intended to be used in lambda expressions, preventing unnecessary code
- Example:

```
books.stream().map(b -> b.getTitle())   ➔
books.stream().map(Book::getTitle)
```

- Lambda parameter list and return type must match the signature of the method

Method references (2)

```
public class MainClass {
   public static void main(String[] args) {
      Function<Integer, Integer> f = MainClass::doSomething;
      System.out.println(f.apply(5));
   }
private static Integer doSomething(Integer amazingInteger) {
      return amazingInteger;
   }
```

Is iterative way better? Real world example

```
Set<Seller> sellers = new HashSet<>();
for(Transaction t : txns) {
     if(t.getBuyer().getAge() > 65) {
          seller.add(t.getSeller());
     }
}
List<Seller> sorted = new ArrayList<>(sellers);
Collections.sort(sorted, new Comperator<Seller>(){
     public int compare(Seller a, Seller b){
          return a.getName().compareTo(b.getName());
     }
}
for(Seller s : sorted){
     System.out.println(s.getName());
}
```

declarative way? Real world example

```
txns.stream()
      .filter(t -> t.getBuyer().getAge() > 65)
      .map(t -> t.getSeller())
      .distinct()
.sorted((s1,s2) ->
            s1.getName().compareTo(s2.getName()))
.forEach(s -> System.out.println(s.getName));
```

Annotations in Java 5/6/7

- Annotations on class declarations

```
@Stateless
public class Person
```

- Annotations on method declarations

```
@Override
public String toString() {
```

- Annotations on class fields

```
@PersistenceContext
private EntityManager em;
```

- Annotations on method parameters

```
public Person getPersonByName(@PathParam("name")
    String name) {
```

New in Java 8

You can put annotations anywhere a type is specified:

```
public void sayHello() {
    @Encrypted String data;
    List<@NonNull String> strings;
    HashMap names = (@Immutable
        HashMap) map;
}
```

Gotchas

- Nobody will stop you doing bad things:

```
public void passPassword(
                @Encrypted String pwd) {…}
public void hello() {
        @PlainText String myPass = "foo";
        passPassword(myPass);
    }
```

- The code above will compile, run… and …crash

Also you cant override based on Annotations

```
interface Test {
   public void sayHello(@NotNull String notNull);
   public void sayHelloNullable(String canNull);
}
```

```
class TestImpl implements Test {
   @Override
   public void sayHello(String notNull) {
      // TODO Auto-generated method stub
   }
   @Override
   public void sayHelloNullable(@NotNull String notNull) {
      // TODO Auto-generated method stub
   }
}
```

Method parameter names

- Before Java 8, only parameter positions and types were preserved after compilation
- In Java 8 you can have those in the .class files
- They are not available by default
  - Need to enable it with `-parameters` option to javac
- And yes you can access this with Reflection…

Exact Numeric Operations

- Java 8 has added several new “exact” methods to the Math class(and not only) protecting sensitive code from implicit overflows

```
int safeC = Math.multiplyExact(bigA, bigB);
// will throw ArithmeticException if result
exceeds +-2^31
```

Exact Numeric Operations(2)

4 new methods added to BigInteger class (also to BigDecimal)
longValueExact(),
intValueExact(),
shortValueExact(),
and byteValueExact().
and toBigIntegerExact() for BigDecimal
All of the newly introduced “xxxxxExact()” methods throw an ArithmeticException if the number contained in the BigInteger instance cannot be provided in the specified form without loss of information

Nashorn

Java(Script)

Was ist das?

- Oracles runtime for ECMAScript 5.1
- GPL licensed
- Part of OpenJDK
- Just type jjs in the shell… and you are in!

Why ?

- Atwoods law: any application that *can* be written in JavaScript, *will* eventually be written in JavaScript
- Oracle proving ground for support of dynamic languages
- Currently supports ECMAScript 5.1 (100%). No backwards compatibility.
- Why not Rhino ? All code compiled to bytecode. No interpretation.

Java -> JavaScript

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class EvalScript {
    public static void main(String[] args) throws Exception {
        // create a script engine manager
        ScriptEngineManager factory = new ScriptEngineManager();
        // create a Nashorn script engine
        ScriptEngine engine = factory.getEngineByName("nashorn");
        // evaluate JavaScript statement
        try {
            engine.eval("print('Hello, World!');");
        } catch (final ScriptException se) { se.printStackTrace();
}
    }
}
```

Multiple Invoke

```
import javax.script.Invocable;
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;

public class Eval_Invocable {
    public static void main(String[] args) throws Exception {
        ScriptEngineManager factory = new ScriptEngineManager();
        ScriptEngine engine = factory.getEngineByName("nashorn");

        engine.eval("function sum(x,y){return x+y}");

        Invocable i = (Invocable) engine;

        System.out.println(i.invokeFunction("sum",5,10));
        System.out.println(i.invokeFunction("sum",10,15));
    }
}
```

Implement Interface

```
public interface Adder {
    int sum(int a,int b);
}
…
ScriptEngine engine = new
ScriptEngineManager().getEngineByName("nashorn");
engine.eval("var i = 5;\n" +
        "function run(){print('runnable' + i++)}");

engine.eval("function sum(x,y){return x+y}");

Invocable i = (Invocable) engine;
Runnable r = i.getInterface(Runnable.class);
r.run();
Adder adder= i.getInterface(Adder.class);
System.out.println(adder.sum(1,3));
```

JavaScript -> Java

```
var timer = new java.util.Timer();

timer.schedule(
   new java.util.TimerTask({
        run: function(){
            print("Tick")
        }
   })
   ,0,1000)
java.lang.Thread.sleep(5000)
timer.cancel();
```

Construct Java(Script) objects

```
var linkedList = new java.util.LinkedList()

var LinkedList = java.util.LinkedList
var list = new LinkedList()

var LinkedList = Java.type(“java.util.LinkedList”)
var list = new LinkedList()
```

Types

```
var ints = new (Java.type(“int[]”))(6)
ints[0]=1
ints[1]=1.6
ints[2]=null
ints[3]=“45”
ints[4]=“str”
Ints[5]=undefined
print(ints)
print(java.util.Arrays.toString(ints))

Output will be:
[I@43re2sd
[1, 1, 0, 45, 0, 0]
```

Types(2)

```
var dbls = new (Java.type(“double[]”))(6)
dbls [0]=1
dbls [1]=1.6
dbls [2]=null
dbls [3]=“45”
dbls [4]=“str”
dbls [5]=undefined
print(dbls )
print(java.util.Arrays.toString(dbls ))

Output will be:
[D@43re2sd
[1.0, 1.6, 0.0, 45.0, NaN, NaN]
```

Type conversion

- Passing JavaScript values to Java methods will use all allowed Java method invocation conversions… + all allowed JavaScript conversions

- All native JS objects implement java.util.Map

- And they do not implement java.util.List

Type conversion(2)

```
MyJavaClass.about(123);
// class java.lang.Integer
MyJavaClass.about(49.99);
// class java.lang.Double
MyJavaClass.about(true);
// class java.lang.Boolean
MyJavaClass.about("hi there")
// class java.lang.String
MyJavaClass.about(new Number(23));
// class jdk.nashorn.internal.objects.NativeNumber
MyJavaClass.about(new Date());
// class jdk.nashorn.internal.objects.NativeDate
MyJavaClass.about(new RegExp());
// class jdk.nashorn.internal.objects.NativeRegExp
MyJavaClass.about({foo: 'bar'});
// class jdk.nashorn.internal.scripts.JO4
```

Arrays conversion

- Not converted automatically!
- Explicit APIs provided:

```
var javaArray = Java.toJavaArray(jsArray,type)
var jsArray = Java.toJavaScriptArray(javaArray)
```

JavaScript lambdas

```
var stack = new java.util.LinkedList();
[1, 2, 3, 4,54,87,42,32,65,4,5,8,43].forEach(function(item) {
  stack.push(item);
});


var sorted = stack
  .stream()
  .filter(function(i){return i%2==0})
  .sorted()
  .toArray();
print(java.util.Arrays.toString(sorted));
```

Output:

```
[2, 4, 4, 8, 32, 42, 54]
```

Scripting extensions

- Additional classpath elements can be specified for the Java Virtual Machine (JVM).
- JavaScript strict mode can be activated.
- An intriguing *scripting mode* can be enabled.

Shell invocations

```
var lines =
`ls -lsa`.split("\n");
for each (var line in lines) {
  print("|> " + line);
}
```

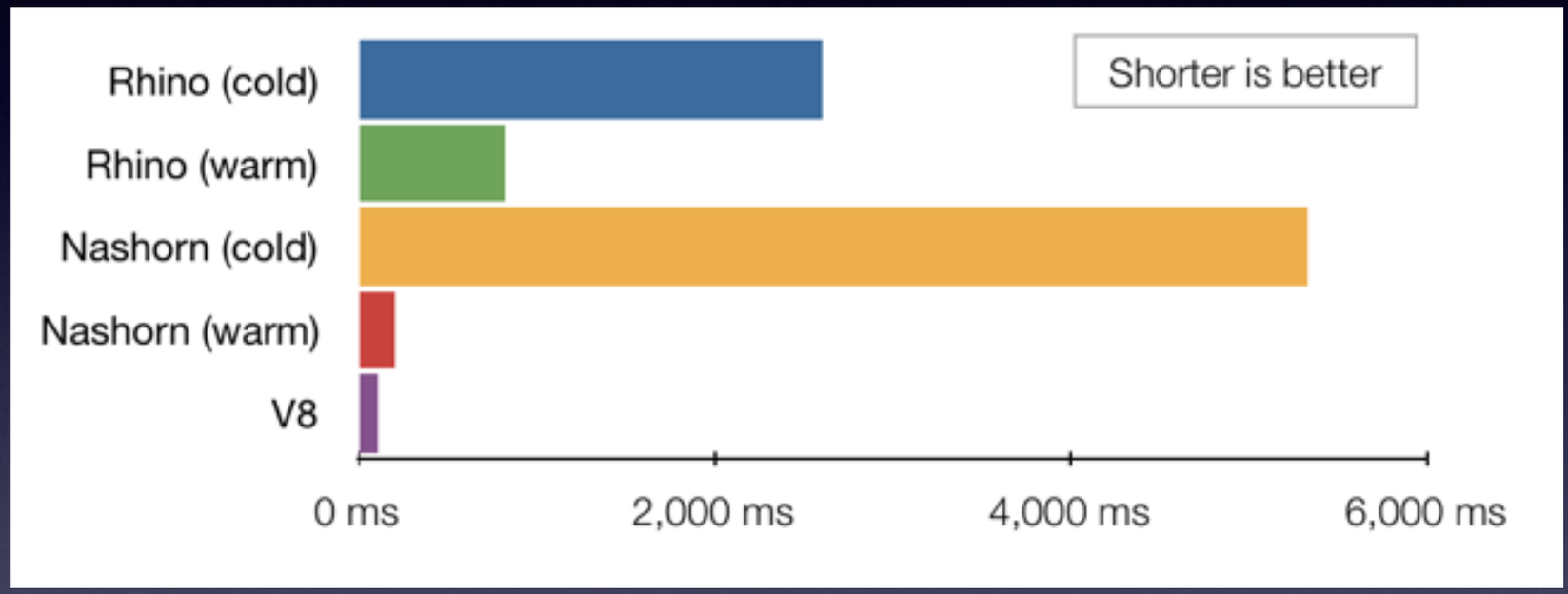
```
Naydens-MacBook-Pro:SoftUniConf 2014 gochev$ jjs -scripting scripting.js
|> total 5328
|>    0 drwxrwxrwx  11 gochev  staff      374 Sep 24 22:09 .
|>    0 drwxrwxrwx  12 gochev  staff      408 Sep 22 18:03 ..
|>   16 -rw-r--r--@  1 gochev  staff     6148 Sep 24 00:15 .DS_Store
|>    0 drwxrwxrwx   7 gochev  staff      238 Sep 22 16:59 0-BigIntegerDemo
|>    0 drwxrwxrwx   7 gochev  staff      238 Sep 22 16:59 1-FirstEvenBlaBla
```

Shell invocations(2)

```
#!/usr/bin/env jjs -scripting
print("Arguments (${$ARG.length})");
for each (arg in $ARG) {
  print("- ${arg}")
}

$ chmod +x executable.js
$ ./executable.js
Arguments (0)
$ ./executable.js -- hello world !
Arguments (3)
- hello
- world
- !
```

Benchmark


Rhino (cold)
Rhino (warm)
Nashorn (cold)
Nashorn (warm)
V8
Shorter is better
0 ms
2,000 ms
4,000 ms
6,000 ms

Do we still have time ? no really ? RLY ?

OK so "How Lambda magic work?"…

Are they like anonymous classes ?

What we(well… ok I) didn't speak about? …

- Date and Time Api
- Stamped Locks
- Concurrent Adders
- Parallel Streams showing why the streaming API is so f*cking awesome
- JavaFX
- Secure Random generation
- AND A LOT MORE

(check this out www.baeldung.com/java8)

That's all Folks!

Contacts

Blog : http://gochev.org
Facebook: https://www.facebook.com/gochev
Linkedin: https://www.linkedin.com/in/gochev
Skype: joke.gochev

Oh yes .. also http://dotnethater.blogspot.com