Ten Things You Can Do With GraalVM

Hands-On Lab

Chris Seaton
Research Manager
Oracle Labs
October 2018

ORACLE CODE

Live for the Code

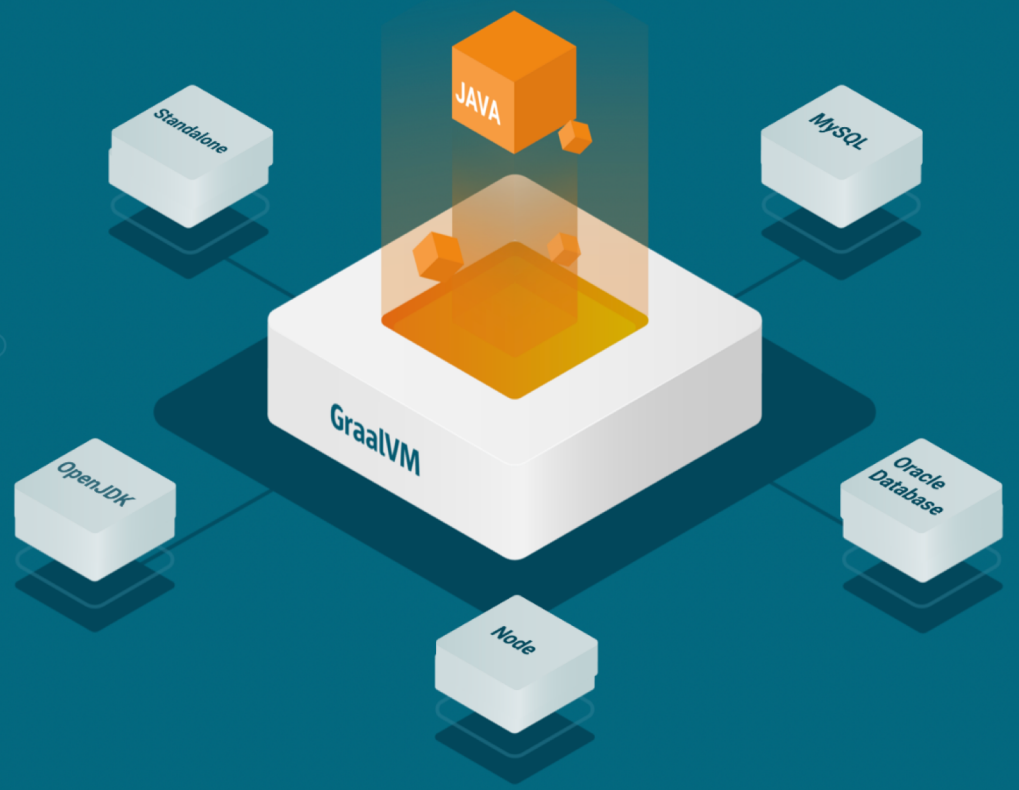
ORACLE

Safe Harbor Statement

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

GraalVM™

Run Programs Faster Anywhere



- What, in concrete terms, is GraalVM?
- What can I practically do with it?
- What do all these things have to do with each other?
- What is the big idea?
- What kind of change is this going to enable?

- What, in concrete terms, is GraalVM?
- What can I practically do with it?
- What do all these things have to do with each other?
- What is the big idea?
- What kind of change is this going to enable?

- Explain through demos
- We'll just dive in and figure out what it all means as we go
- Can follow on if you want, but can also just watch me
- Some demos easier to follow than others

What we need to get started

Files you'll need

- You'll need GraalVM to do anything – https://graalvm.org/
- Plus some source files to run on it
- Download everything:
  - https://www.dropbox.com/s/1t2ninff60ya4ni/ten-things-graalvm.tar.gz (https://goo.gl/K14jfa)
- Download just the sources to run:
  - https://www.dropbox.com/s/o7064h37wmdpbbs/ten-things-graalvm-sources.tar.gz (https://goo.gl/b3ncW4)
- I also have these on USB drives for an emergency

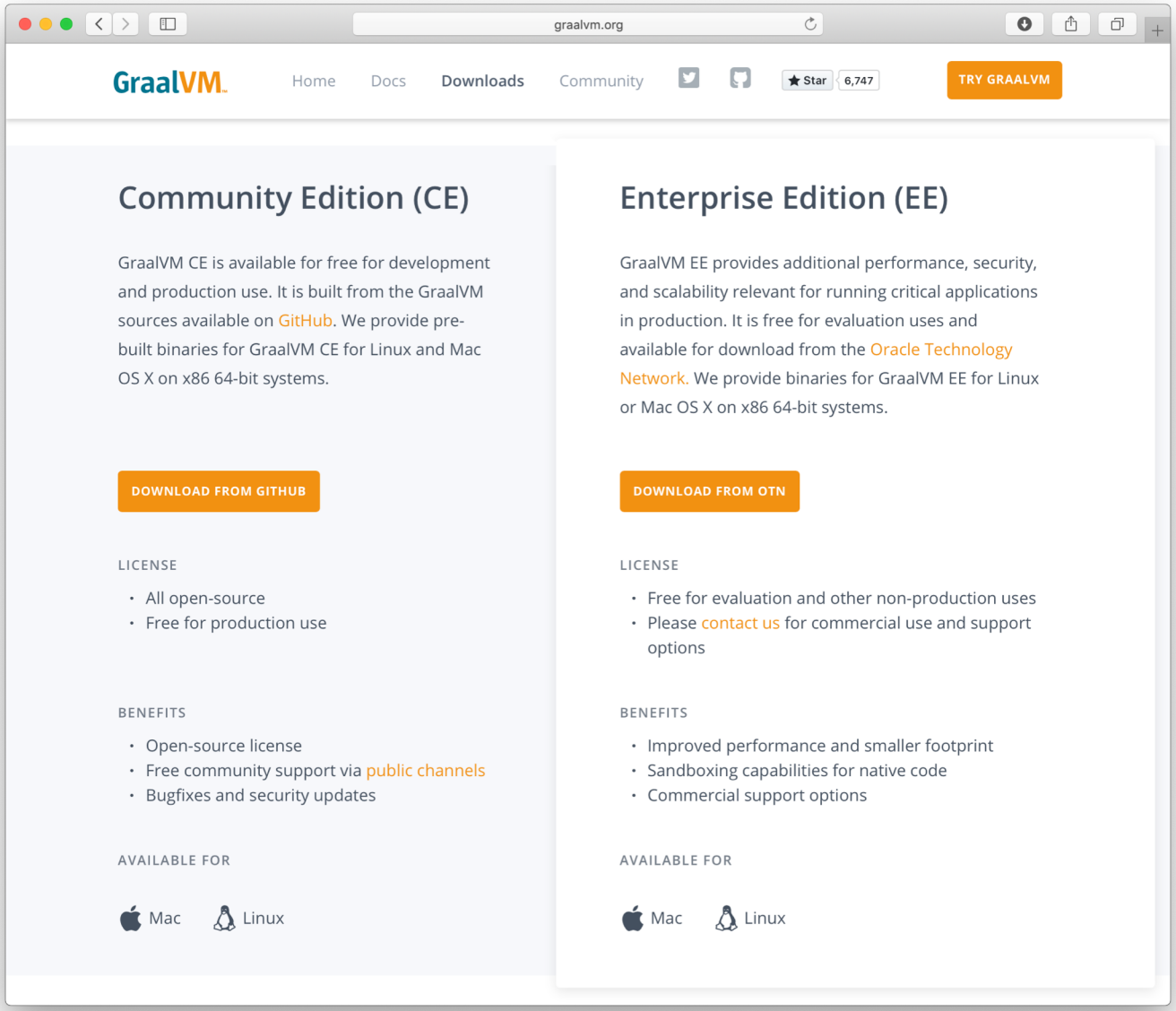
Files you'll need

- Some demos use some other common tools – Maven, Google Chrome
- Some use some more complex software – Docker, Oracle Database

- But feel free to just watch if you want! I'll run the demos myself

System you'll need

- AMD64
- macOS or Linux



High performance Java

ORACLE

Using GraalVM as your JDK

- You can use GraalVM as a drop-in replacement for OpenJDK
- 1.8 at the moment, will be updated to the 11 LTS soon
- Includes all the same commands, flags, options and so on

Add it to your $PATH

```
$ export PATH=graalvm-ee-1.0.0-rc8/Contents/Home/bin:$PATH
```

```
public static void main(String[] args) {
    Arrays.stream(args)
            .flatMap(TopTen::fileLines)
            .flatMap(line -> Arrays.stream(line.split("\\b")))
            .map(word -> word.replaceAll("[^a-zA-Z]", ""))
            .filter(word -> word.length() > 0)
            .map(word -> word.toLowerCase())
            .collect(Collectors.groupingBy(Function.identity(), Collectors.counting()))
            .entrySet().stream()
            .sorted((a, b) -> -a.getValue().compareTo(b.getValue()))
            .limit(10)
            .forEach(e -> System.out.format("%s = %d%n", e.getKey(), e.getValue()));
}
```

ORACLE

Compile and run as normal

```
$ javac TopTen.java
$ time java TopTen large.txt
…
real 0m18.905s
```

Compare to standard OpenJDK

```
$ time java -XX:-UseJVMCICompiler TopTen large.txt
…
real 0m23.102s
```

What is going on?

- The Graal just-in-time compiler is one part of GraalVM
- It replaces (or runs as a tier above) the existing JIT compilers like C2
- It's written in Java, which we think lets us improve it more easily, so it achieves better performance than C2
- Here we're getting 20% faster performance on a benchmark
- Twitter see 18% faster in production on real Scala applications, using only the CE version – EE not needed for high performance
- Is it odd that a JIT compiler for Java is written in Java?

```
for (MonitorExitNode monitorExitNode : graph.getNodes(MonitorExitNode.TYPE)) {
    FixedNode next = monitorExitNode.next();
    if ((next instanceof MonitorEnterNode || next instanceof RawMonitorEnterNode)) {
        // should never happen, osr monitor enters are always direct successors of the graph
        // start
        assert !(next instanceof OSRMonitorEnterNode);
        AccessMonitorNode monitorEnterNode = (AccessMonitorNode) next;
        if (isCompatibleLock(monitorEnterNode, monitorExitNode)) {
            /*
             * We've coarsened the lock so use the same monitor id for the whole region,
             * otherwise the monitor operations appear to be unrelated.
             */
            MonitorIdNode enterId = monitorEnterNode.getMonitorId();
            MonitorIdNode exitId = monitorExitNode.getMonitorId();
            if (enterId != exitId) {
                enterId.replaceAndDelete(exitId);
            }
            GraphUtil.removeFixedWithUnusedInputs(monitorEnterNode);
            GraphUtil.removeFixedWithUnusedInputs(monitorExitNode);
        }
    }
}
```

JVMCI is the interface that lets you plug in a new JIT

```
$ time java -XX:-UseJVMCICompiler TopTen large.txt
…
real 0m23.102s
```

This is one way to use GraalVM

- Just using it as a faster version of the JDK
- Drop-in replacement gives you Graal by default, everything else is unmodified

High performance Java on OpenJDK 11

ORACLE

Graal also works on standard OpenJDK 11

- Graal (the JIT compiler part) is also included in Graal
- As an experimental, unsupported option, hidden behind flags
- Older version, due to the release cycle
- We'd recommend using the GraalVM package to experiment with

Enable Graal in OpenJDK 11

On AMD64, on macOS and Linux

```
$ java -XX:+UnlockExperimentalVMOptions \
       -XX:+EnableJVMCI \
       -XX:+UseJVMCICompiler \
       …
```

Doesn’t mention Graal, does it? JVMCI does service discovery and automatically finds Graal as the only JVMCI compiler available

```
 * ...                                  and/or its affiliates. All rights reserved.
 */

package valhalla;

public value class Complex {

    private final double re;
    private final double im;

    Complex(double re, double im) {
        this.re =  re;
        this.im =  im;
    }

    public double re() { return re; }

    public double im() { return im; }

    public Complex add(Complex that) {
--  Complex.java  /demo/valhalla/src/main/java/valhalla/Complex.java  Top 26:28  (
Switch to buffer (default log): l
```

```java
public class Complex {
  private final double re;
  private final double im;

  public Complex(double re, double im) {
    this.re = re;
    this.im = im;
  }

  public double re() { return re; }
  public double im() { return im; }

  public Complex add(Complex that) {
    return new Complex(this.re + that.re, this.im + that.im);
  }

  public Complex mul(Complex that) {
    return new Complex(this.re * that.re - this.im * that.im,
                       this.re * that.im + this.im * that.re);
  }
}
```

```java
public Complex[][] multiply() {
  int size = A.length;
  Complex[][] R = new Complex[size][size];
  for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
      Complex s = new Complex(0, 0);
      for (int k = 0; k < size; k++) {
        s = s.add(A[i][k].mul(B[k][j]));
           }
           R[i][j] = s;
    }
  }
  return R;
}
```

Compile and run on OpenJDK 11

```
$ mvn package
$ java -jar target/benchmarks.jar -prof gc
```

Run on OpenJDK 11 with Graal enabled

```
$ java -XX:+UnlockExperimentalVMOptions \
       -XX:+EnableJVMCI \
       -XX:+UseJVMCICompiler \
       -jar target/benchmarks.jar -prof gc
```

2.4x faster by enabling Graal

```
valhallaBench.Multiply.multiply  9231.032 us/op
valhallaBench.Multiply.multiply  3774.706 us/op
```

This is a second way to use GraalVM

- Not using GraalVM, but using the key component from it, Graal
- Already present in OpenJDK 11
- Just need to enable it using some flags
- This is effectively what Twitter do (they probably build their own JDK and Graal, not sure, but it’s a detail)

Low footprint, fast startup Java

What about shorter running applications or functions?

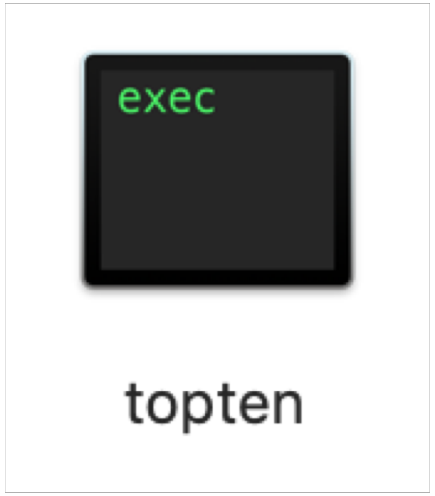
- The JVM typically has a relatively slow time to start
  - Compared to simpler VMs, like Python or Ruby
  - Compared to native executables like those produced from Go or Rust
  - JRuby ‘hello world’ startup time is an order of magnitude worse than standard Ruby
- The JVM typically takes up a relatively large amount of disk space
  - Can be helped with `jlink` – down to tens of MB
- The JVM typically takes up a relatively large amount of RAM
  - Interpreter, compiler, classfile parser, verifier etc all take up space

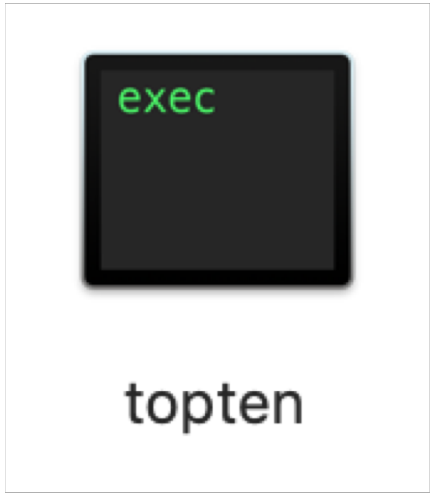
Run as normal

```
$ time java TopTen small.txt
…
real 0m0.408s
```

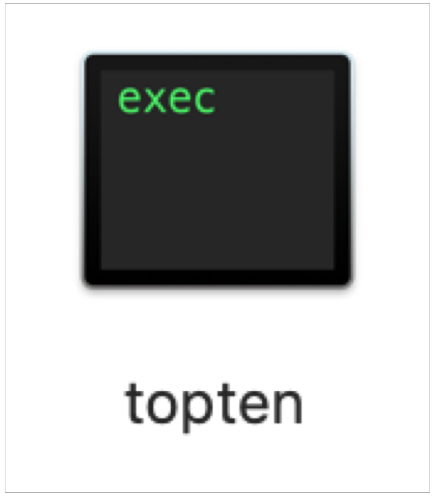
Compile to native using GraalVM

```
$ native-image TopTen
…
$ time ./topten small.txt
…
real 0m0.112s
```

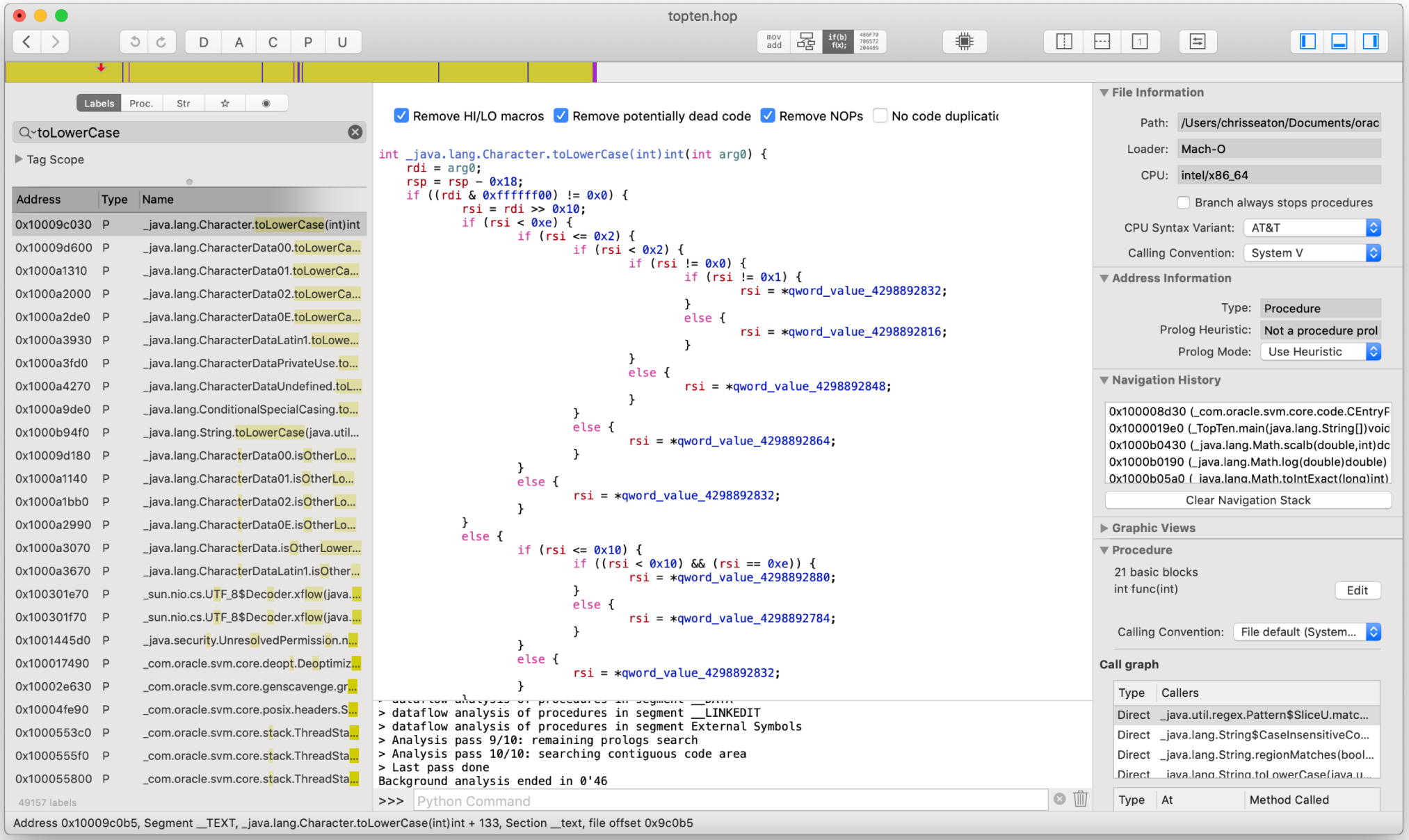





```
$ du -h topten
8.8M topten
```




```
$ otool -L topten
topten:
/usr/lib/libSystem.B.dylib
/usr/lib/libz.1.dylib
/System/Library/CoreFoundation
```



So what does this have to do with a JIT?

- Graal is written in Java
- So it can be used as a library from other Java code
- We realized that we could write a program to use it ahead-of-time, to build and ahead-of-time compiler
- This is what the `native-image` tool is – a Java application that uses Graal as a library

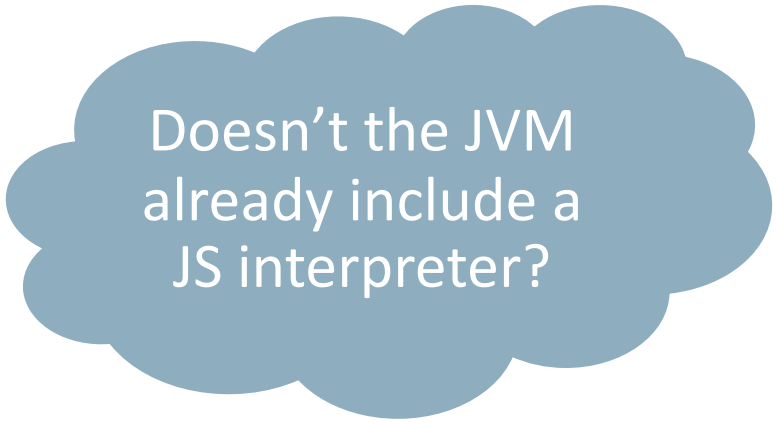
This is a third way to use GraalVM

- Using the `native-image` tool to ahead-of-time compile your application to native code
- No dependency on the JVM
- Small executables – Docker deployments
- Starts quickly, has a low footprint

Run other languages

GraalVM includes a new JavaScript interpreter

```
$ js –version
Graal JavaScript 1.0 (GraalVM CE Native 1.0.0-rc8)

$ js
> print("hello");
hello
```



GraalVM also includes an implementation of Node.js

```
$ node --version
v10.9.0

$ npm --version
6.2.0
```

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('<h1>Hello!</h1>');
});

app.listen(8080, function () {
  console.log('serving at http://localhost:8080')
});
```

```
$ npm install express

$ node hello-express.js
serving at http://localhost:8080
```

You can plug new languages into GraalVM

```
$ gu install ruby
$ gu install python
$ gu install R
```

This version works offline with the files on the USB drives

```
$ gu install --file ruby-installable-ce-1.0.0-rc8-macos-amd64.jar
$ gu install --file python-installable-ce-1.0.0-rc8-macos-amd64.jar
$ gu install --file r-installable-ce-1.0.0-rc8-macos-amd64.jar
```

```
$ gu rebuild-images polyglot libpolyglot js llvm python ruby
```

In reality you'd want to run this, but it takes a long time

You can plug new languages into GraalVM

```
$ ruby --version
truffleruby 1.0.0-rc8, like ruby 2.4.4, GraalVM CE Native [x86_64-darwin]

$ graalpython --version
Graal Python 3.7.0 (GraalVM CE Native 1.0.0-rc8)

$ R --version
R version 3.4.0 (FastR)
```

So what does this have to do with a JIT?

- We realized instead of writing languages that emit bytecode at runtime (JRuby) we could write languages that use the Graal JIT directly
- But that's hard, so we realized we could write a framework, Truffle, to do that automatically, based on a simple interpreter
- Faster, as they use a more powerful JIT more directly
- Simpler, because a framework does most of the hard work, so easy to implement lots of languages
- Interopable (polyglot) because they all use the same system
- Using `native-image` they start quickly

This is a fourth way to use GraalVM

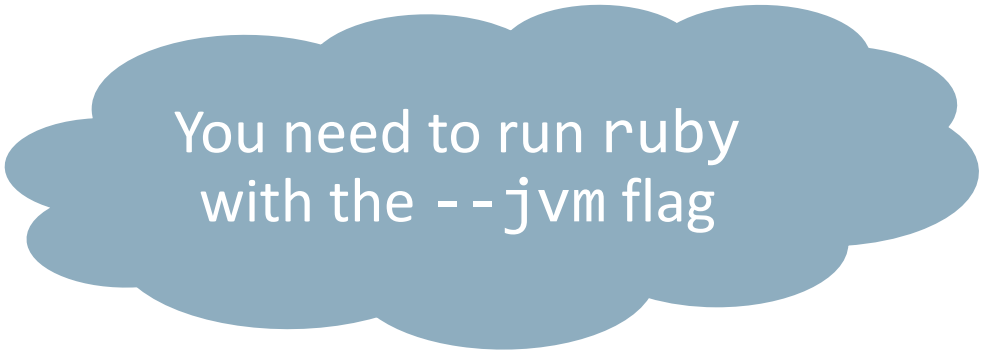
- As a multi language platform
- Use as a drop-in replacement for your existing language platform
- Faster than standard implementations

Polyglot

GraalVM is polyglot as well as multi-language

- Like many languages implemented on the JVM, our languages can use Java libraries
- Run `ruby-java.rb`

```
BigInteger = Java.type('java.math.BigInteger')
puts BigInteger.valueOf(2).pow(100).toString
```



GraalVM is polyglot as well as multi-language

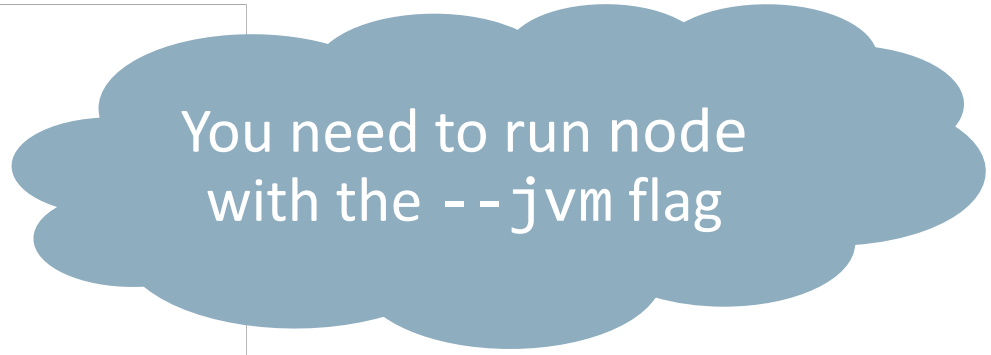
- Run `python-java.rb`

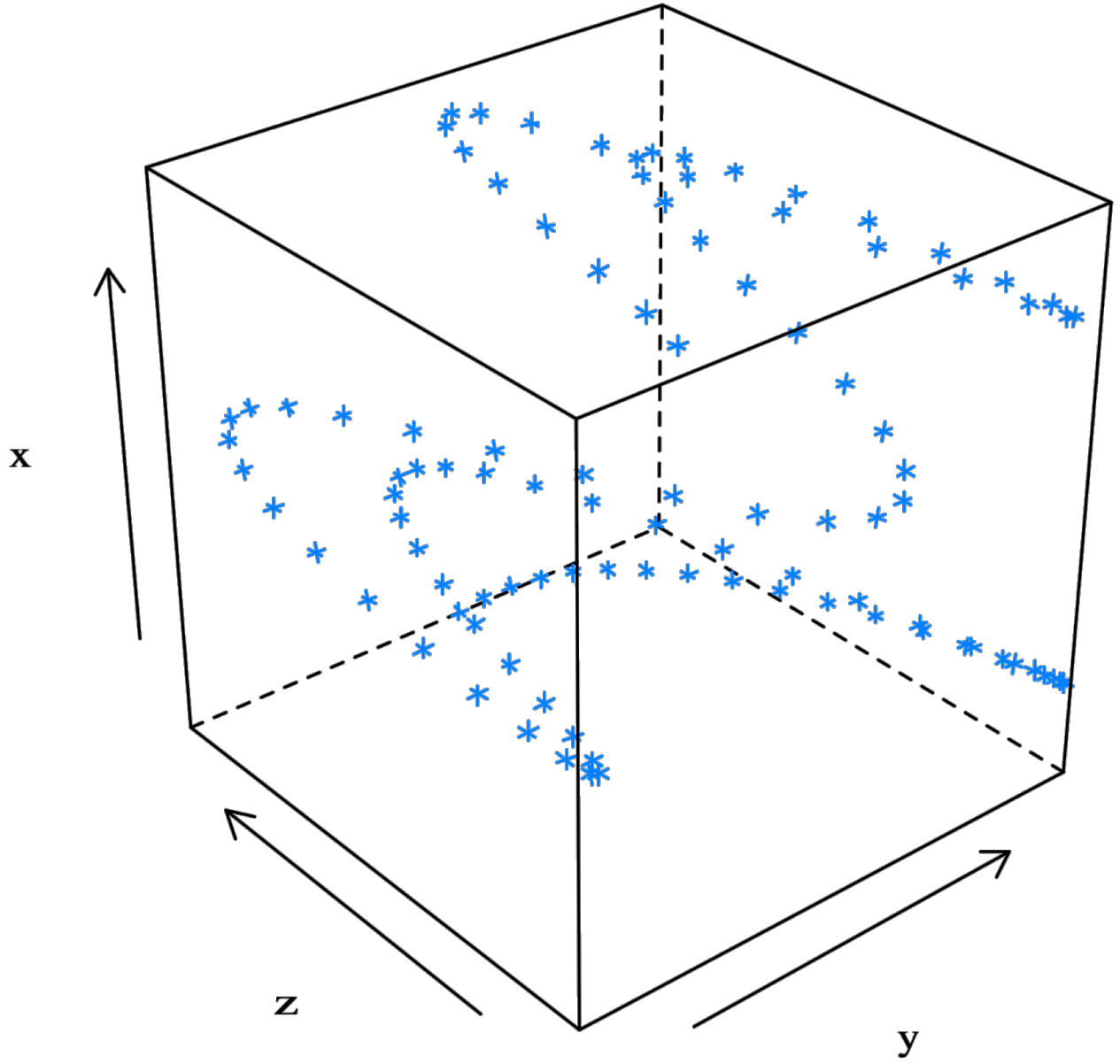
```
import java
BigInteger = java.type('java.math.BigInteger')
print(BigInteger.valueOf(2).pow(100).toString())
```

You need to run `graalpython` with the `--jvm` flag

Java is just another language in this case

```
const express = require('express');
const app = express();

app.get('/', function (req, res) {
  res.send(Interop.eval('R',
    `svg();
     require(lattice);
     x <- 1:100
     y <- sin(x/10)
     z <- cos(x^1.3/(runif(1)*5+10))
     print(cloud(x~y*z, main="cloud plot"))
     grDevices:::svg.off()
    `));
})

app.listen(8080, function () {
  console.log('serving at http://localhost:8080');
});
```





So what does this have to do with a JIT?

- All the languages use the same JIT, and the same high-level implementation framework, so they can all work together
- Integration is at a higher level than with traditional bytecode implementation

This is our fifth way to use GraalVM

- As a polyglot language platform
- Java interop
- Poyglot interop

Native languages on the JVM

ORACLE

Is our approach only suited to Java and dynamic languages?

- There's nothing special about native languages
- C has the same `if` statements and `while` loops Ruby does
- C has pointers and malloc, but so does Ruby in its `FFI` module

Is our approach only suited to Java and dynamic languages?

- Example – running `gzip` on the JVM
  - Not a clean piece of code
  - 8.6 k lines of C
  - Macros, pointer arithmetic, unions
  - We'll avoid the complexity of autotools and make by using a single-file version
  - http://people.csail.mit.edu/smcc/projects/single-file-programs/

```
if (more == (unsigned)EOF) {
    /* Very unlikely, but possible on 16 bit machine if strstart == 0
     * and lookahead == 1 (input done one byte at time)
     */
    more--;
} else if (strstart >= WSIZE+MAX_DIST) {
    /* By the IN assertion, the window is not empty so we can't confuse
     * more == 0 with more == 64K on a 16 bit machine.
     */
    Assert(window_size == (ulg)2*WSIZE, "no sliding with BIG_MEM");

    memcpy((char*)window, (char*)window+WSIZE, (unsigned)WSIZE);
    match_start -= WSIZE;
    strstart    -= WSIZE; /* we now have strstart >= MAX_DIST: */
   if (rsync_chunk_end != 0xFFFFFFFFUL)
       rsync_chunk_end -= WSIZE;

    block_start -= (long) WSIZE;

    for (n = 0; n < HASH_SIZE; n++) {
        m = head[n];
        head[n] = (Pos)(m >= WSIZE ? m-WSIZE : NIL);
    }
    for (n = 0; n < WSIZE; n++) {
        m = prev[n];
        prev[n] = (Pos)(m >= WSIZE ? m-WSIZE : NIL);
        /* If n is not on any hash chain, prev[n] is garbage but
         * its value will never be used.
         */
    }
    more += WSIZE;
}
```

```
$ clang -c -emit-llvm gzip.c
$ gzip small.txt
$ lli gzip.bc -d small.txt.gz
```

So what does this have to do with a JIT?

- We can use the JIT that we use for Java, JavaScript, Ruby, Python, R and so on, for C as well
- Actually – any language that can target LLVM
- C, C++, Objective C, Swift, Fortran, Rust, etc
- Genuine potential for dynamic optimization
- Potential for sandboxing as well

This is our sixth way to use GraalVM

- To run native applications on the JVM
- Our demo showed a full application
- More probably use-case is running native libraries from Java or other managed languages

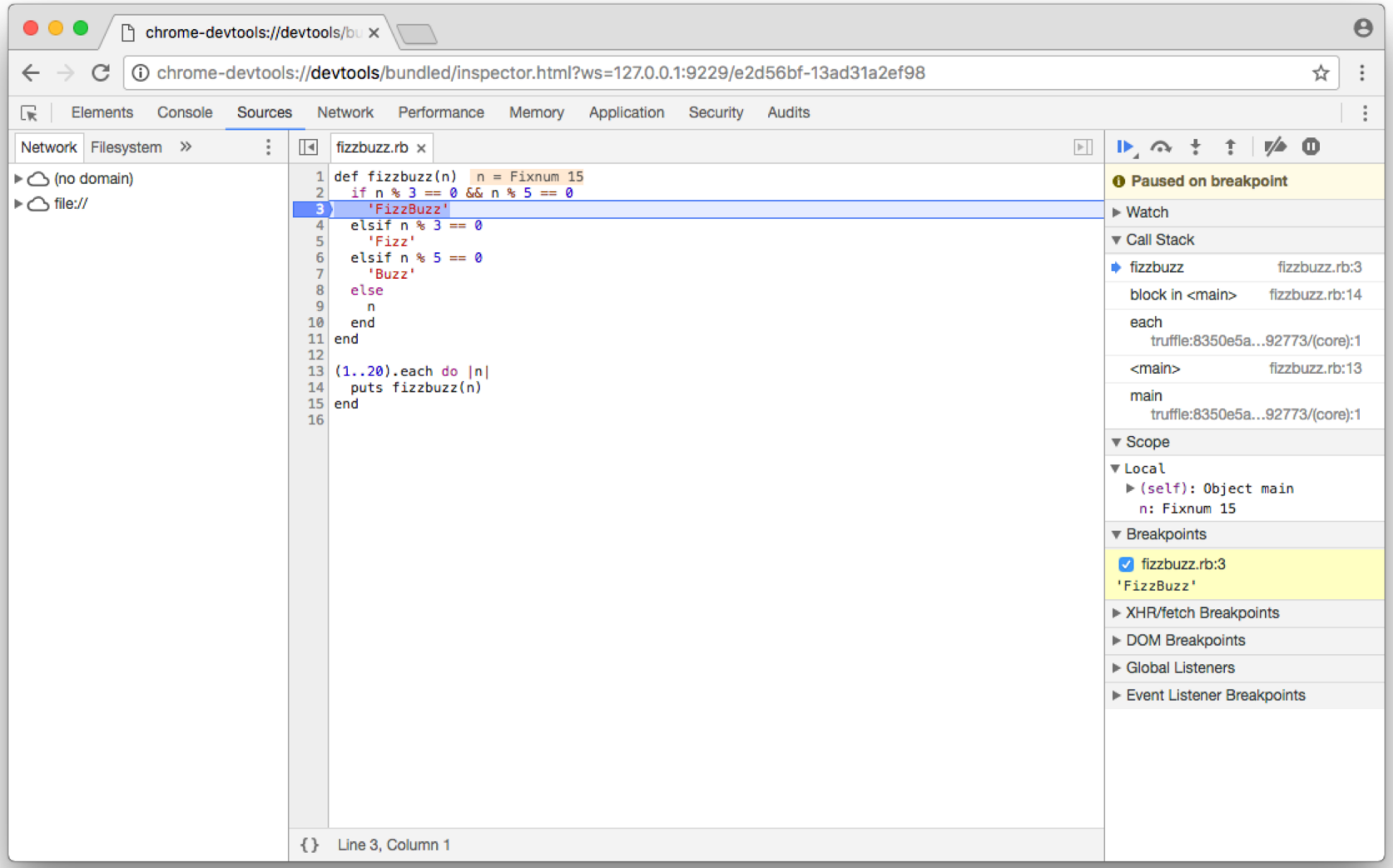
Debugging

Tooling for these extra languages

- With other languages on the JVM you usually have to use a Java debugger
- Perhaps with source information in the bytecode for the guest language
- Some custom debuggers, but not for all languages

```ruby
def fizzbuzz(n)
  if n % 3 == 0 && n % 5 == 0
    'FizzBuzz'
  elsif n % 3 == 0
    'Fizz'
  elsif n % 5 == 0
    'Buzz'
  else
    n
  end
end

(1..20).each do |n|
  puts fizzbuzz(n)
end
```

```
$ ruby fizzbuzz.rb
…
$ ruby --inspect fizzbuzz.rb
Debugger listening on port 9229.
To start debugging, open the following URL in Chrome:
    chrome-devtools://devtools/bundled/…
```



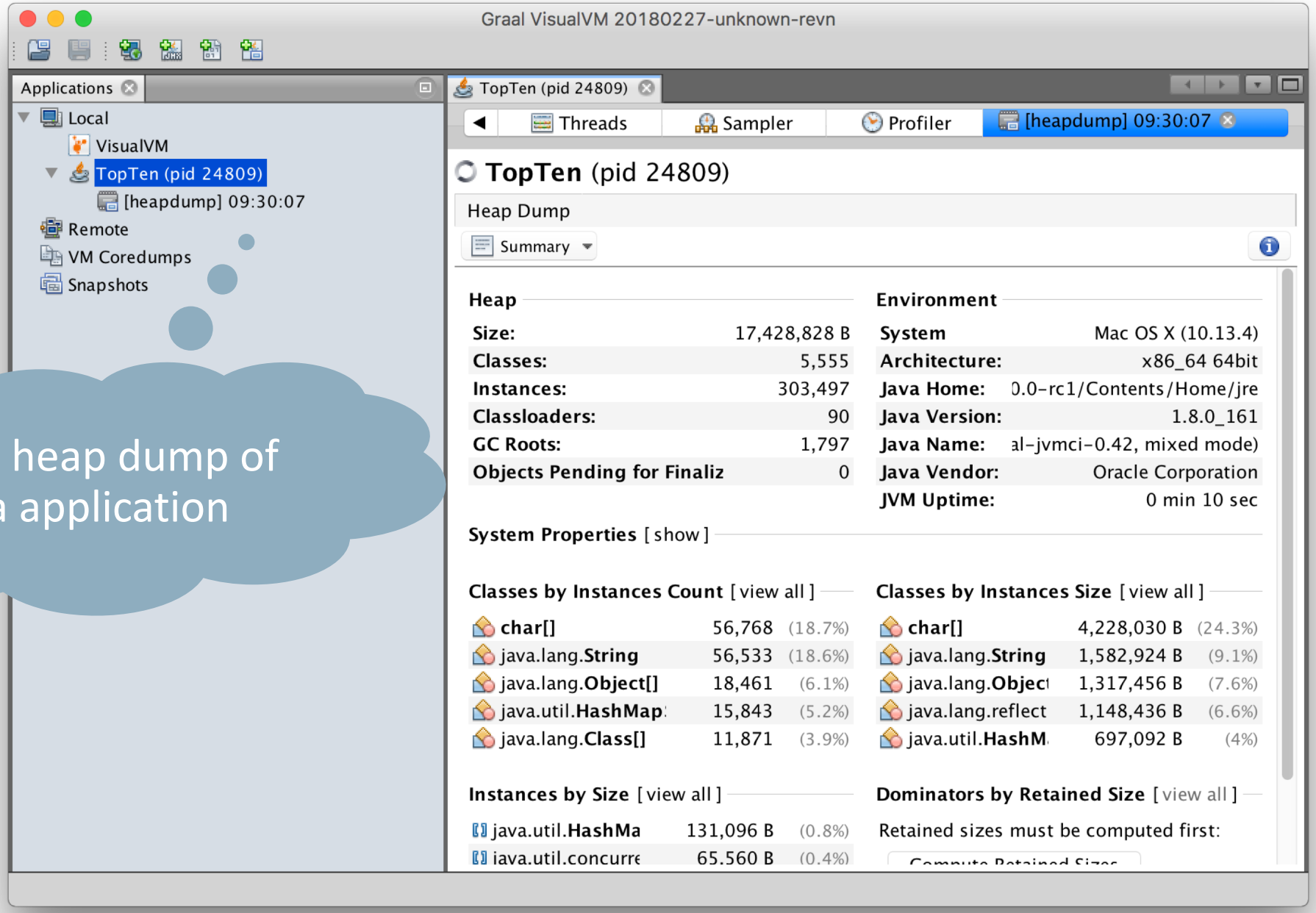
So what does this have to do with a JIT?

- All the languages are implemented in the same framework, so the debugger can understand them all via that framework
- The Graal JIT has support for deoptimization, so can debug optimized code running in production

This is our seventh way to use GraalVM

- As a way to get a debugger for multiple languages
- Not all languages have a community large enough to support a high quality debugger
- Few have a user interface as nice as Chrome
- The debugger works cross-language as well
- If you implement your own language on our framework, you get this debugger for free

Monitoring

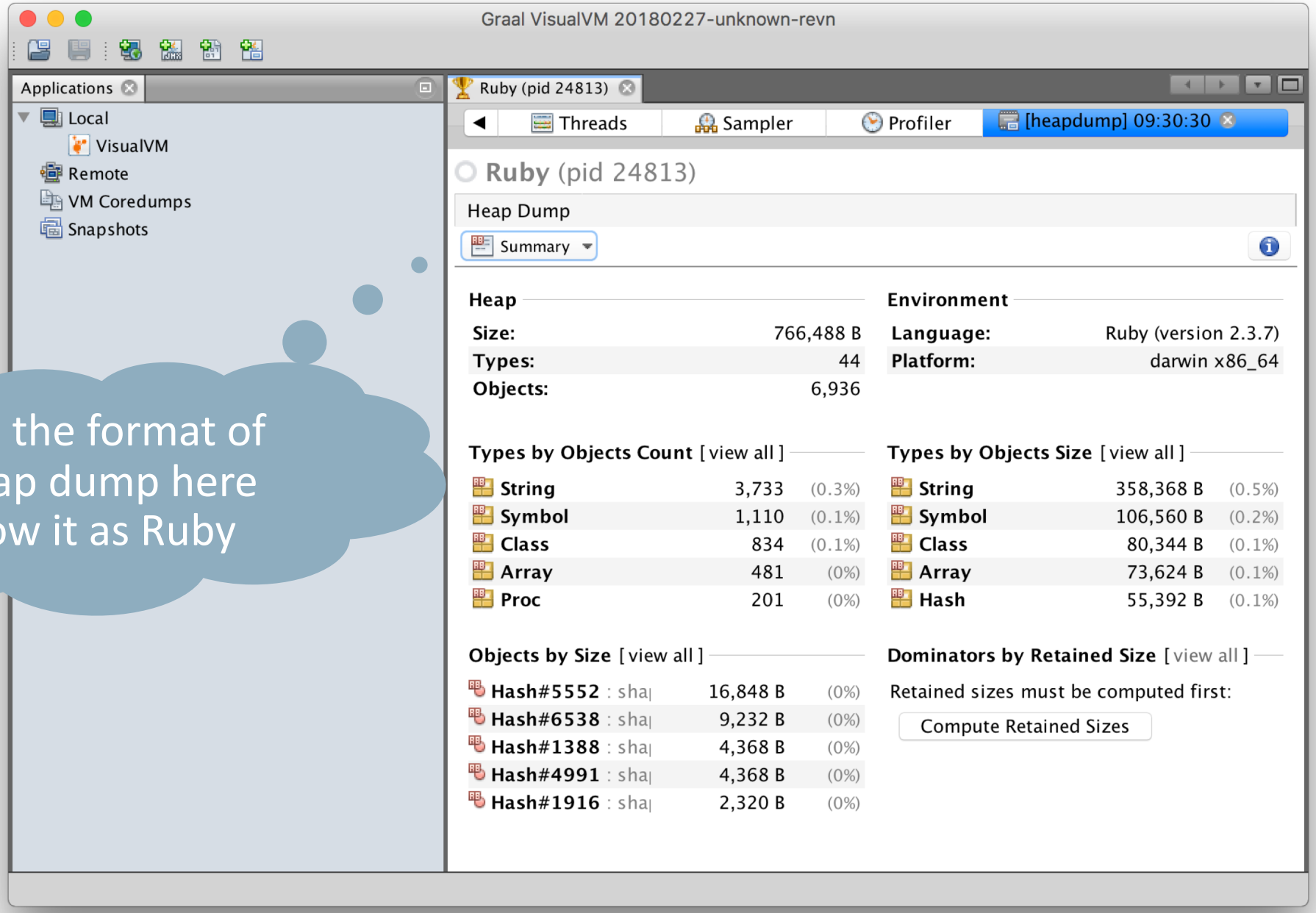


Taking a heap dump of a Java application

Monitoring for these extra languages

- VisualVM and other similar tools let you monitor the JVM
- Non-JVM languages often don’t have this kind of tool
- With other languages on the JVM, then often show the underlying Java objects, rather than the guest language objects

```
$ ruby render.rb

$ jvisualvm
```



Change the format of the heap dump here to show it as Ruby

So what does this have to do with a JIT?

- Like the debugger, all languages use the same framework so VisualVM can understand them all
- All the languages use the same system to implement their objects on top of Java objects

This is our eighth way to use GraalVM

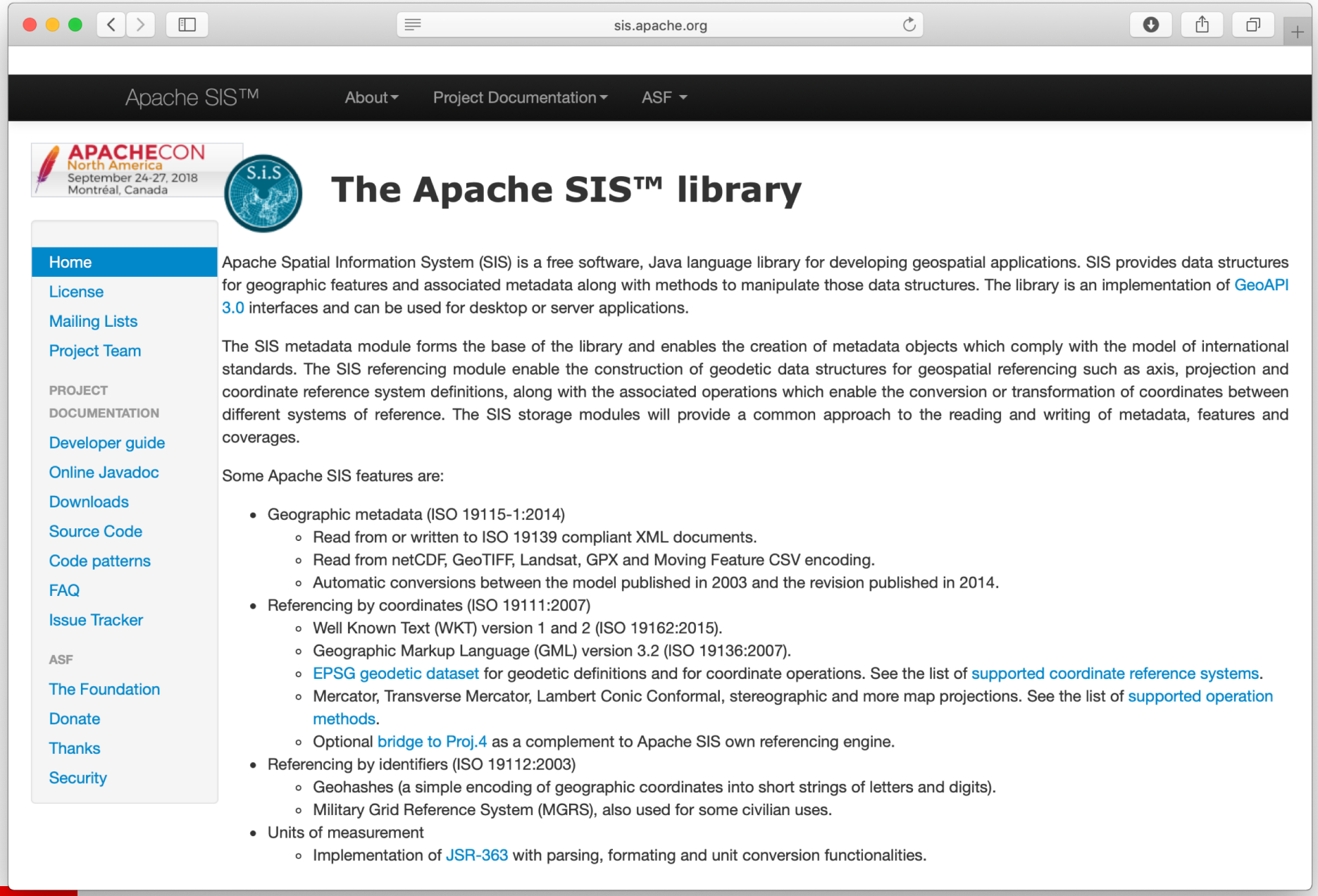
- Like the debugger, to get high quality tooling for other languages

Java as a native library

ORACLE

Java code as a native library

- The Java ecosystem is phenomenal
- Often more and better libraries than available in other languages
- In the examples so far, it's always been the Java code that has owned the process
- Can we run Java code inside another application that we already have?



```java
import org.apache.sis.distance.DistanceUtils;

public class Distance {

    public static void main(String[] args) {
        final double aLat   = Double.parseDouble(args[0]);
        final double aLong  = Double.parseDouble(args[1]);
        final double bLat   = Double.parseDouble(args[2]);
        final double bLong  = Double.parseDouble(args[3]);
        System.out.printf("%f km%n", DistanceUtils.getHaversineDistance(aLat, aLong, bLat, bLong));
    }

}
```

```
$ javac -cp sis.jar -parameters Distance.java
$ java -cp sis.jar:. Distance \
      51.507222 -0.1275 40.7127 -74.0059
5570.25 km
```

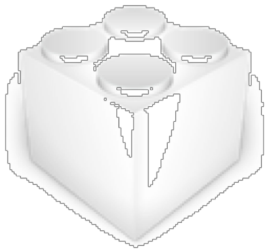
```
$ native-image -cp sis.jar:. Distance
...
$ ./distance 51.507222 -0.1275 40.7127 -74.0059
5570.25 km
```

```
...
import org.graalvm.nativeimage.IsolateThread;
import org.graalvm.nativeimage.c.function.CEntryPoint;

public class Distance {

    ...

    @CEntryPoint(name = "distance")
    public static double distance(IsolateThread thread,
          double a_lat, double a_long,
          double b_lat, double b_long) {
        return DistanceUtils.getHaversineDistance(a_lat, a_long, b_lat, b_long);
    }

    ...

}
```

```
$ native-image -cp sis.jar:. -H:Kind=SHARED_LIBRARY \
       -H:Name=libdistance
```

```c
#include <stdlib.h>
#include <stdio.h>

#include <libdistance.h>

int main(int argc, char **argv) {
  graal_isolate_t *isolate = NULL;
  graal_isolatethread_t *thread = NULL;

  if (graal_create_isolate(NULL, &isolate) != 0 || (thread = graal_current_thread(isolate)) == NULL) {
    fprintf(stderr, "initialization error\n");
    return 1;
  }

  double a_lat   = strtod(argv[1], NULL);
  double a_long  = strtod(argv[2], NULL);
  double b_lat   = strtod(argv[3], NULL);
  double b_long  = strtod(argv[4], NULL);

  printf("%f km\n", distance(thread, a_lat, a_long, b_lat, b_long));

  return 0;
}
```

```
$ clang -I. -L. -ldistance distance.c -o distance
$ otool -L distance
distance:
	libdistance.dylib
	/usr/lib/libSystem.B.dylib
$ ./distance 51.507222 -0.1275 40.7127 -74.0059
5570.25 km
```



libdistance.dylib

This is our ninth way to use GraalVM

- To compile Java code to native libraries
- Can then embed in a native application
- Or any language with a native FFI (Ruby, Python, Rust, Haskell, Go, etc)

Polyglot in the database

Demo using the Oracle Database MLE

- Multi-lingual (polyglot) edition
- Available as a Docker image
- Subject to the Oracle Technology Network license agreement, so you need to accept that and download it yourself

https://oracle.github.io/oracle-db-mle/releases/0.2.7/docker/

```
$ docker load --input mle-docker-0.2.7.tar.gz # takes a while

$ docker run mle-docker-0.2.7 # takes a while
$ docker ps
$ docker exec -ti <container_id> bash -li
```

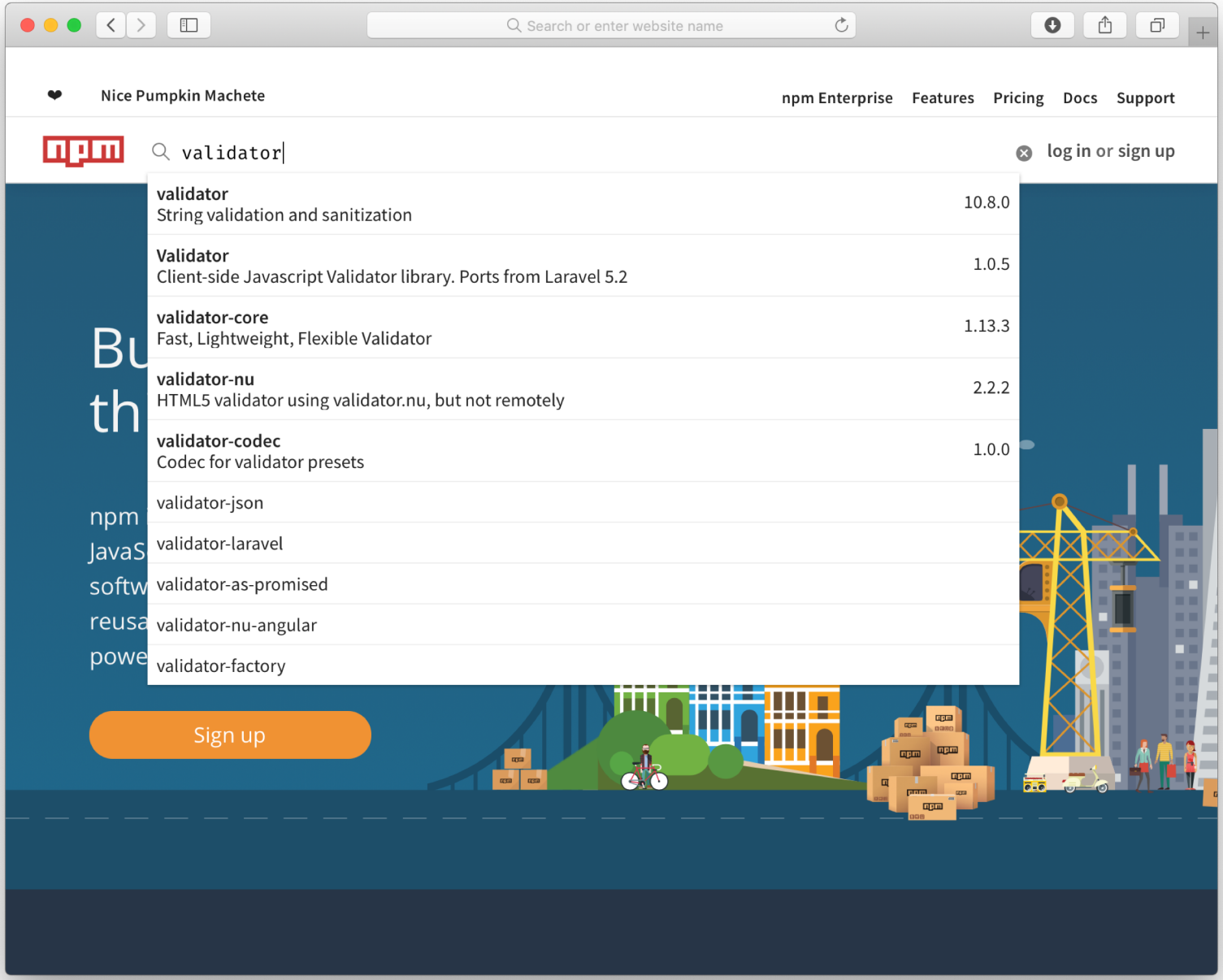
JavaScript in the client and frontend, Oracle in the backend

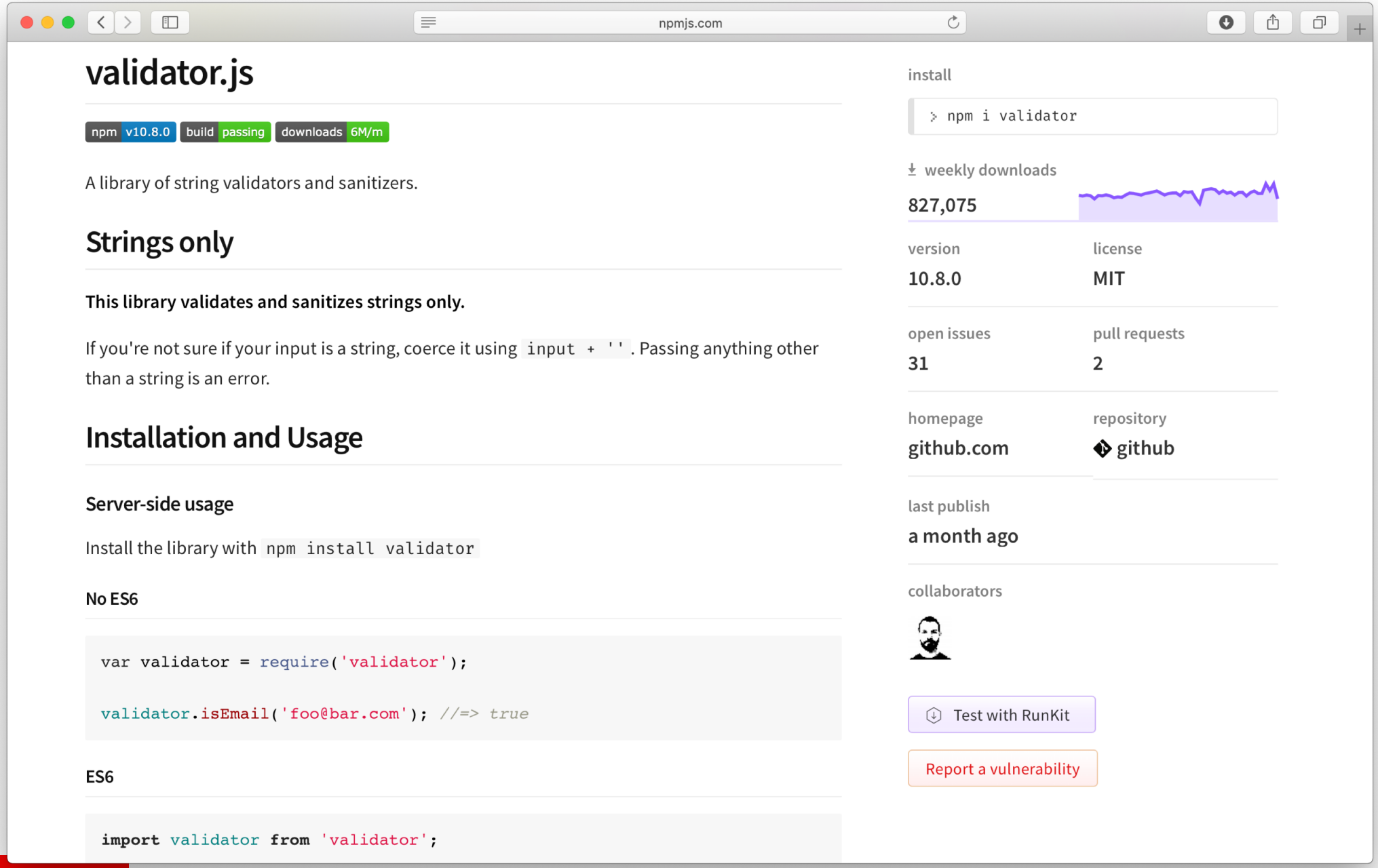
JS

JS

ORACLE DATABASE

ORACLE





```
$ echo "{}" > package.json
$ npm install validator
$ npm install @types/validator

$ dbjs deploy -u scott -p tiger -c localhost:1521/ORCLCDB validator

$ sqlplus scott/tiger@localhost:1521/ORCLCDB
```

```
SQL> select validator.isEmail('oleg.selaev@oracle.com') from dual;
SQL> select validator.isEmail('oleg.selaev') from dual;
```

This is our tenth and final way to use GraalVM

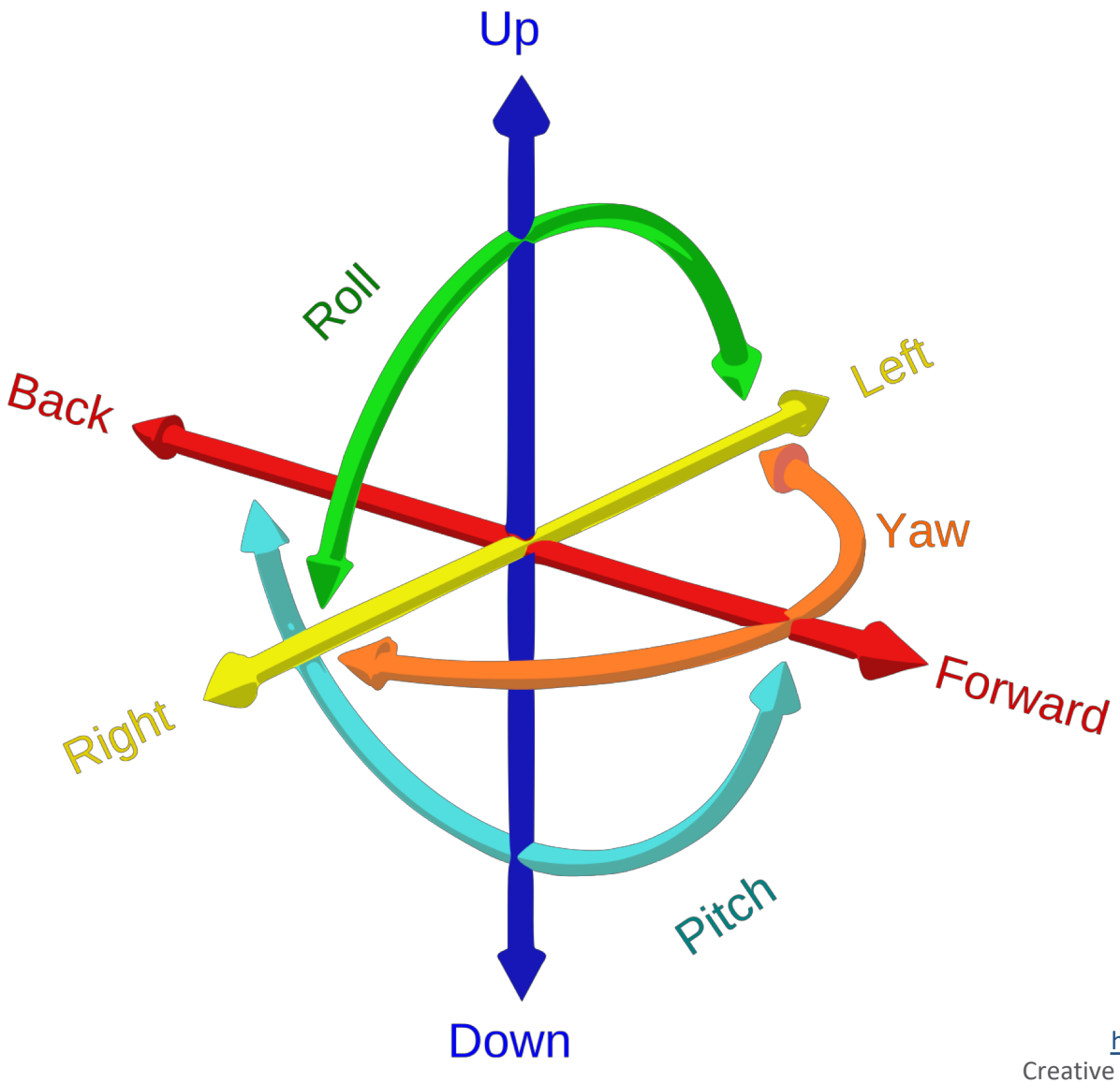
- To run user functions (stored procedures) in the Oracle Database
- JavaScript at the moment, but Python soon, can also demo Ruby and R
- The same idea works in MySQL

- Note this functionality isn't part of GraalVM

How does this bring it all together?

- This is a…
  - JavaScript interpreter, implemented in Java, using our framework
  - Using the polyglot interface to talk to the query language
  - Compiled ahead-of-time using Graal into a native library
  - Which can be linked into the database
  - Including Graal as a JIT within that library for high performance

Wrap up



https://commons.wikimedia.org/wiki/File:6DOF.svg
Creative Commons Attribution-Share Alike 4.0 International

I think it’s about giving people degrees of freedom

- Let people run the language they want
- With the ecosystem of libraries they want
- On the JVM or on native
- Embedded or embedding
- With the tooling they want
- With the performance they want
- ‘One compiler to rule them all’

Get in touch with us

- https://graalvm.org/
- https://twitter.com/ChrisGSeaton
- https://gitter.im/graalvm/graal-core

Team

Oracle
Florian Angerer
Danilo Ansaloni
Stefan Anzinger
Martin Balin
Cosmin Basca
Daniele Bonetta
Dušan Bálek
Matthias Brantner
Lucas Braun
Petr Chalupa
Jürgen Christ
Laurent Daynès
Gilles Duboscq
Svatopluk Dědic
Martin Entlicher
Pit Fender
Francois Farquet
Brandon Fish
Matthias Grimmer
Christian Häubl
Peter Hofer
Bastian Hossbach
Christian Humer
Tomáš Hůrka
Mick Jordan

Oracle (continued)
Vojin Jovanovic
Anantha Kandukuri
Harshad Kasture
Cansu Kaynak
Peter Kessler
Duncan MacGregor
Jiří Maršík
Kevin Menard
Miloslav Metelka
Tomáš Myšík
Petr Pišl
Oleg Pliss
Jakub Podlešák
Aleksandar Prokopec
Tom Rodriguez
Roland Schatz
Benjamin Schlegel
Chris Seaton
Jiří Sedláček
Doug Simon
Štěpán Šindelář
Zbyněk Šlajchrt
Boris Spasojevic
Lukas Stadler
Codrut Stancu

Oracle (continued)
Jan Štola
Tomáš Stupka
Farhan Tauheed
Jaroslav Tulach
Alexander Ulrich
Michael Van De Vanter
Aleksandar Vitorovic
Christian Wimmer
Christian Wirth
Paul Wögerer
Mario Wolczko
Andreas Wöß
Thomas Würthinger
Tomáš Zezula
Yudi Zheng

Red Hat
Andrew Dinn
Andrew Haley

Intel
Michael Berg

Twitter
Chris Thalinger

Oracle Interns
Brian Belleville
Ondrej Douda
Juan Fumero
Miguel Garcia
Hugo Guiroux
Shams Imam
Berkin Ilbeyi
Hugo Kapp
Alexey Karyakin
Stephen Kell
Andreas Kunft
Volker Lanting
Gero Leinemann
Julian Lettner
Joe Nash
Tristan Overney
Aleksandar Pejovic
David Piorkowski
Philipp Riedmann
Gregor Richards
Robert Seilbeck
Rifat Shariyar

Oracle Alumni
Erik Eckstein
Michael Haupt
Christos Kotselidis
David Leibs
Adam Welc
Till Westmann

JKU Linz
Hanspeter Mössenböck
Benoit Daloze
Josef Eisl
Thomas Feichtinger
Josef Haider
Christian Huber
David Leopoldseder
Stefan Marr
Manuel Rigger
Stefan Rumzucker
Bernhard Urban

TU Berlin:
Volker Markl
Andreas Kunft
Jens Meiners
Tilmann Rabl

University of Edinburgh
Christophe Dubach
Juan José Fumero Alfonso
Ranjeet Singh
Toomas Remmelg

LaBRI
Floréal Morandat

University of California, Irvine
Michael Franz
Yeoul Na
Mohaned Qunaibit
Gulfem Savrun Yeniceri
Wei Zhang

Purdue University
Jan Vitek
Tomas Kalibera
Petr Maj
Lei Zhao

T. U. Dortmund
Peter Marwedel
Helena Kotthaus
Ingo Korb

University of California, Davis
Duncan Temple Lang
Nicholas Ulle

University of Lugano, Switzerland
Walter Binder
Sun Haiyang

Safe Harbor Statement

The preceding is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.