

Ten Things You Can Do With GraalVM

Hands-On Lab

Chris Seaton
Research Manager
Oracle Labs
October 2018



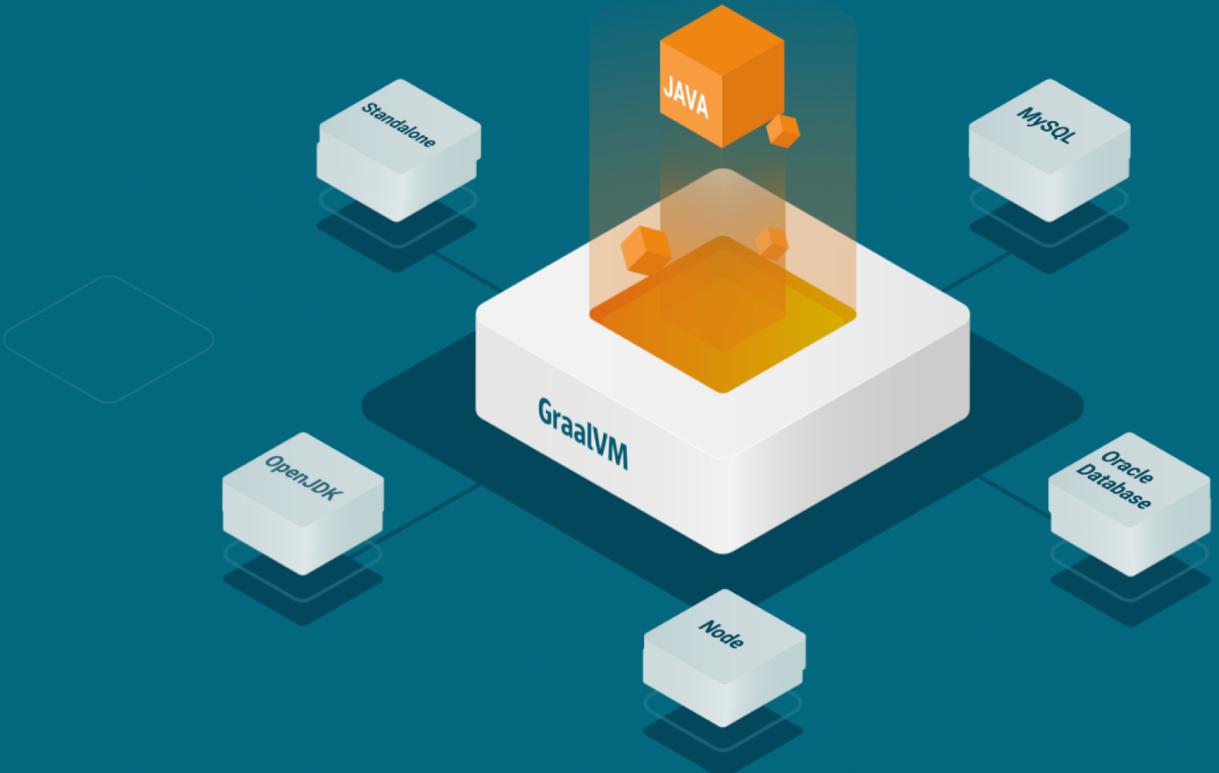
**Live for
the Code**

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

The screenshot shows a web browser window displaying the GraalVM website at graalvm.org. The page is split into two main sections: "Community Edition (CE)" on the left and "Enterprise Edition (EE)" on the right.

Community Edition (CE)

- GraalVM CE is available for free for development and production use. It is built from the GraalVM sources available on [GitHub](#). We provide pre-built binaries for GraalVM CE for Linux and Mac OS X on x86 64-bit systems.
- [DOWNLOAD FROM GITHUB](#)
- LICENSE**
 - All open-source
 - Free for production use
- BENEFITS**
 - Open-source license
 - Free community support via [public channels](#)
 - Bugfixes and security updates
- AVAILABLE FOR**
 - Mac
 - Linux

Enterprise Edition (EE)

- GraalVM EE provides additional performance, security, and scalability relevant for running critical applications in production. It is free for evaluation uses and available for download from the [Oracle Technology Network](#). We provide binaries for GraalVM EE for Linux or Mac OS X on x86 64-bit systems.
- [DOWNLOAD FROM OTN](#)
- LICENSE**
 - Free for evaluation and other non-production uses
 - Please [contact us](#) for commercial use and support options
- BENEFITS**
 - Improved performance and smaller footprint
 - Sandboxing capabilities for native code
 - Commercial support options
- AVAILABLE FOR**
 - Mac
 - Linux



High performance Java

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()));
}
```

Compile and run as normal

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

This demo is
run with the
EE version

Compare to standard OpenJDK

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



I will explain
this flag
shortly...

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



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

*/ ...
* Oracle 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): 1)



```
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);  
    }  
}
```

```
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
```



This is using
Java 11, not
GraalVM!

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

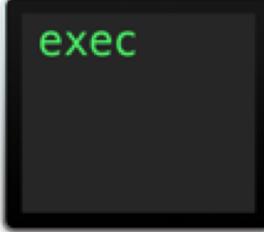
```
$ time java TopTen small.txt
```

...

```
real 0m0.408s
```

Compile to native using GraalVM

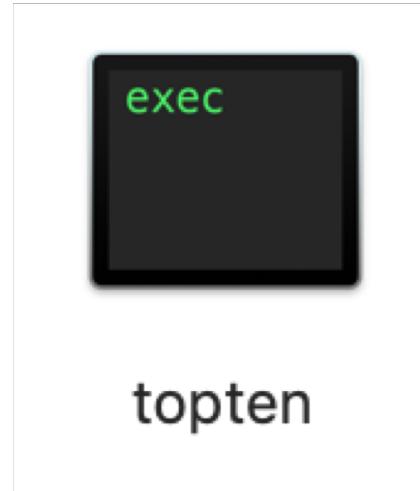
```
$ native-image TopTen  
...  
$ time ./topten small.txt  
...  
real 0m0.112s
```



topten



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



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

topten.hop

Labels Proc. Str ⌂
mov add if(b) f(t); 486F78 786572 284469

Labels Proc. Str ⌂
File Information

Remove HI/LO macros Remove potentially dead code Remove NOPs No code duplication

```

int _java.lang.Character.toLowerCase(int)int(int arg0) {
    rdi = arg0;
    rsp = rsp - 0x18;
    if ((rdi & 0xfffff00) != 0x0) {
        rsi = rdi >> 0x10;
        if (rsi < 0xe) {
            if (rsi <= 0x2) {
                if (rsi != 0x0) {
                    if (rsi != 0x1) {
                        rsi = *qword_value_4298892832;
                    } else {
                        rsi = *qword_value_4298892816;
                    }
                } else {
                    rsi = *qword_value_4298892848;
                }
            } else {
                rsi = *qword_value_4298892864;
            }
        } else {
            rsi = *qword_value_4298892832;
        }
    } else {
        if (rsi <= 0x10) {
            if ((rsi < 0x10) && (rsi == 0xe)) {
                rsi = *qword_value_4298892880;
            } else {
                rsi = *qword_value_4298892784;
            }
        } else {
            rsi = *qword_value_4298892832;
        }
    }
}

```

> dataflow analysis of procedures in segment __DATA
> dataflow analysis of procedures in segment __LINKEDIT
> dataflow analysis of procedures in segment External Symbols
> Analysis pass 9/10: remaining prologs search
> Analysis pass 10/10: searching contiguous code area
> Last pass done
Background analysis ended in 0'46

>>> Python Command

Address 0x10009c0b5, Segment __TEXT, _java.lang.Character.toLowerCase(int)int + 133, Section __text, file offset 0x9c0b5

Path: /Users/chrisseaton/Documents/orac...

Loader: Mach-O

CPU: intel/x86_64

Branch always stops procedures

CPU Syntax Variant: AT&T

Calling Convention: System V

Type: Procedure

Prolog Heuristic: Not a procedure pro...

Prolog Mode: Use Heuristic

Address Information

Navigation History

0x100008d30 (_com.oracle.svm.core.code.CEntryF...
0x1000019e0 (_TopTen.main([java.lang.String[]])voic...
0x1000b0430 (_java.lang.Math.scalb(double,int)dc...
0x1000b0190 (_java.lang.Math.log(double)double)...
0x1000b05a0 (_java.lang.Math.toIntExact(long)int)...
Clear Navigation Stack

Graphic Views

Procedure

21 basic blocks
int func(int)

Calling Convention: File default (System...)

Call graph

Type	Callers
Direct	_java.util.regex.Pattern\$SliceU.match(...) _java.lang.String\$CaseInsensitiveCo... _java.lang.String.regionMatches(bool...) _java.lang.String.toLowerCase(iava.u...)

Type	At	Method Called
------	----	---------------

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
```

Doesn't the JVM
already include a
JS interpreter?

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

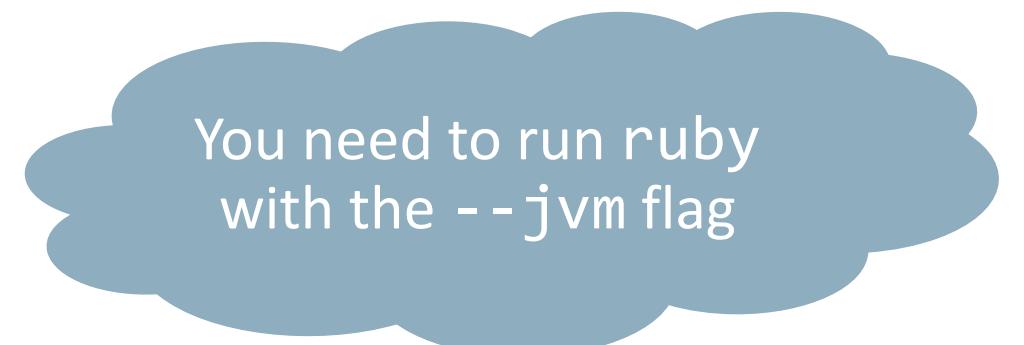
ORACLE®

Copyright © 2018, Oracle and/or its affiliates. All rights reserved. |

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
```



You need to run ruby
with the --jvm flag

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

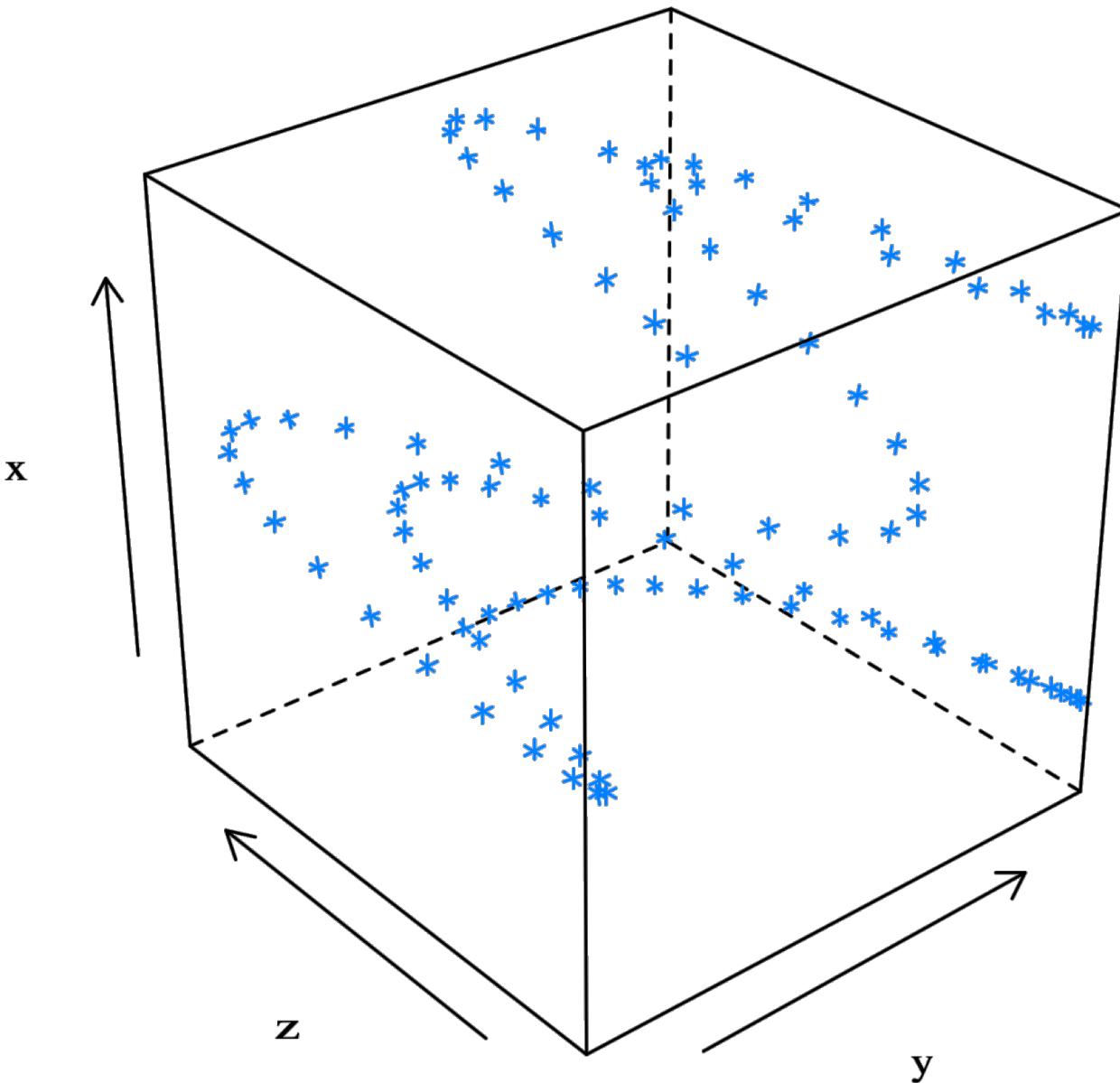
Java is just another language in this case

```
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');
});
```

You need to run node
with the --jvm flag

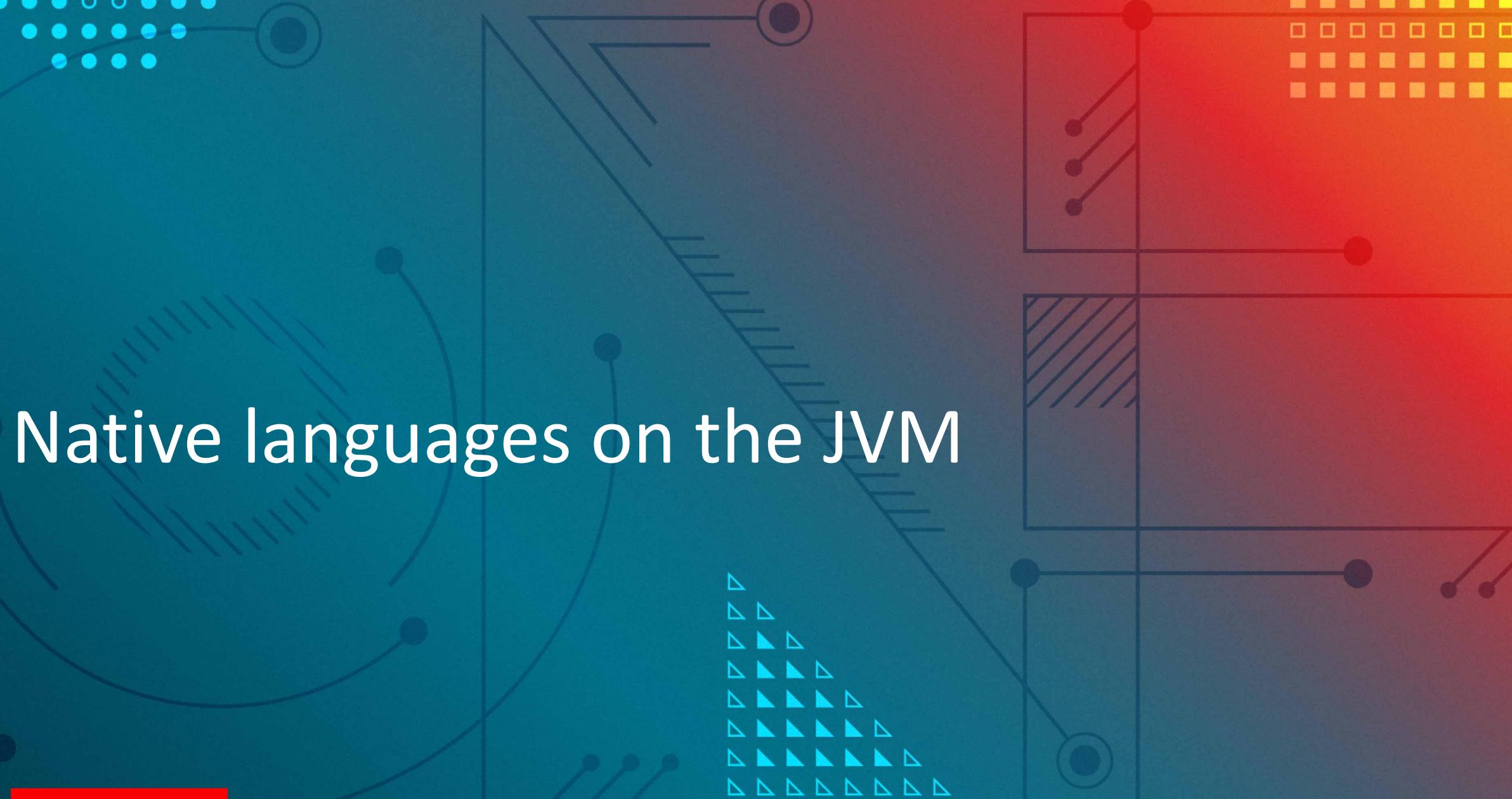


So what does this have to do with a JIT?

- 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
- Polyglot interop



Native languages on the JVM

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 != 0xFFFFFFFFFUL)
        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

```
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/...`

The screenshot shows the Chrome DevTools Sources panel with the file `fizzbuzz.rb` open. The code defines a `fizzbuzz` method that prints numbers from 1 to 20, replacing multiples of 3 with 'Fizz' and multiples of 5 with 'Buzz'. A breakpoint is set at line 3, which contains the string `'FizzBuzz'`. The DevTools interface includes a sidebar with navigation links like Network, Filesystem, and a Call Stack, and a bottom status bar indicating the current line and column.

```
1 def fizzbuzz(n) n = Fixnum 15
2   if n % 3 == 0 && n % 5 == 0
3     'FizzBuzz'
4   elsif n % 3 == 0
5     'Fizz'
6   elsif n % 5 == 0
7     'Buzz'
8   else
9     n
10  end
11 end
12
13 (1..20).each do |n|
14   puts fizzbuzz(n)
15 end
16
```

{ } Line 3, Column 1

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

Graal VisualVM 20180227-unknown-revn

Applications

- Local
 - VisualVM
 - TopTen (pid 24809)
 - [heapdump] 09:30:07
- Remote
- VM Coredumps
- Snapshots

TopTen (pid 24809)

Threads Sampler Profiler [heapdump] 09:30:07

TopTen (pid 24809)

Heap Dump

Summary

Heap

Size:	17,428,828 B
Classes:	5,555
Instances:	303,497
Classloaders:	90
GC Roots:	1,797
Objects Pending for Finaliz	0

Environment

System	Mac OS X (10.13.4)
Architecture:	x86_64 64bit
Java Home:	0.0-rc1/Contents/Home/jre
Java Version:	1.8.0_161
Java Name:	al-jvmci-0.42, mixed mode)
Java Vendor:	Oracle Corporation
JVM Uptime:	0 min 10 sec

System Properties [show]

Classes by Instances Count [view all]

char[]	56,768 (18.7%)
java.lang.String	56,533 (18.6%)
java.lang.Object[]	18,461 (6.1%)
java.util.HashMap	15,843 (5.2%)
java.lang.Class[]	11,871 (3.9%)

Classes by Instances Size [view all]

char[]	4,228,030 B (24.3%)
java.lang.String	1,582,924 B (9.1%)
java.lang.Object	1,317,456 B (7.6%)
java.lang.reflect	1,148,436 B (6.6%)
java.util.HashM	697,092 B (4%)

Instances by Size [view all]

java.util.HashMa	131,096 B (0.8%)
java.util.concurre	65.560 B (0.4%)

Dominators by Retained Size [view all]

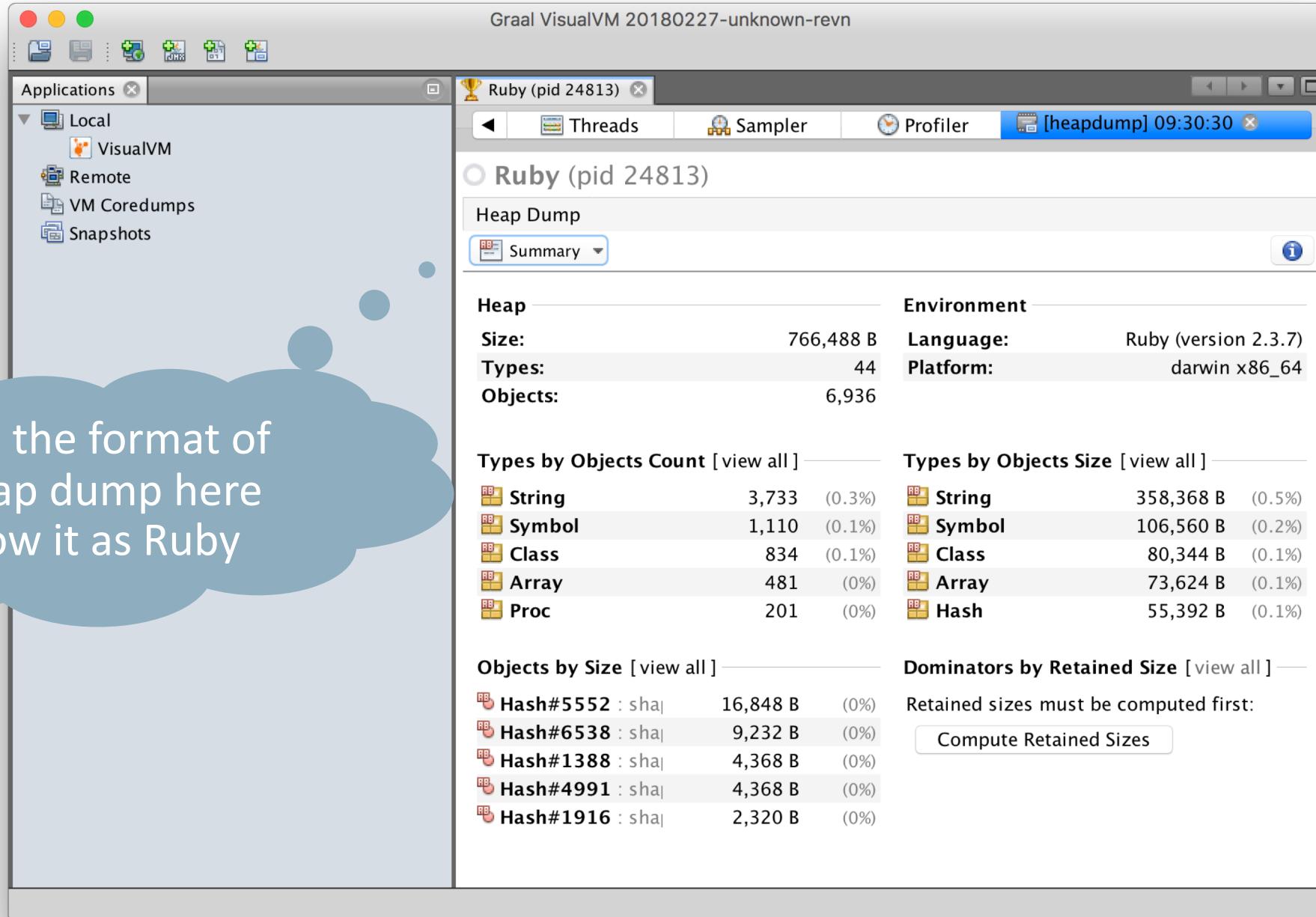
Retained sizes must be computed first:
[Compute Retained Sizes](#)

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
```



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

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?

sis.apache.org

The Apache SIS™ library

Apache Spatial Information System (SIS) is a free software, Java language library for developing geospatial applications. SIS provides data structures for geographic features and associated metadata along with methods to manipulate those data structures. The library is an implementation of [GeoAPI 3.0](#) interfaces and can be used for desktop or server applications.

The SIS metadata module forms the base of the library and enables the creation of metadata objects which comply with the model of international standards. The SIS referencing module enable the construction of geodetic data structures for geospatial referencing such as axis, projection and coordinate reference system definitions, along with the associated operations which enable the conversion or transformation of coordinates between different systems of reference. The SIS storage modules will provide a common approach to the reading and writing of metadata, features and coverages.

Some Apache SIS features are:

- Geographic metadata (ISO 19115-1:2014)
 - Read from or written to ISO 19139 compliant XML documents.
 - Read from netCDF, GeoTIFF, Landsat, GPX and Moving Feature CSV encoding.
 - Automatic conversions between the model published in 2003 and the revision published in 2014.
- Referencing by coordinates (ISO 19111:2007)
 - Well Known Text (WKT) version 1 and 2 (ISO 19162:2015).
 - Geographic Markup Language (GML) version 3.2 (ISO 19136:2007).
 - [EPSG geodetic dataset](#) for geodetic definitions and for coordinate operations. See the list of [supported coordinate reference systems](#).
 - Mercator, Transverse Mercator, Lambert Conic Conformal, stereographic and more map projections. See the list of [supported operation methods](#).
 - Optional [bridge to Proj.4](#) as a complement to Apache SIS own referencing engine.
- Referencing by identifiers (ISO 19112:2003)
 - Geohashes (a simple encoding of geographic coordinates into short strings of letters and digits).
 - Military Grid Reference System (MGRS), also used for some civilian uses.
- Units of measurement
 - Implementation of [JSR-363](#) with parsing, formating and unit conversion functionalities.

```
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
```

```
#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;

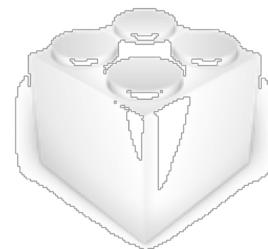
    if (graal_create_isolate(NULL, &isolate) != 0 || (thread = graal_current_thread(isolate)) == NULL) {
        fprintf(stderr, "initialization error\n");
        return 1;
    }

    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



ORACLE®
D A T A B A S E

Search or enter website name

Nice Pumpkin Machete

npm Enterprise Features Pricing Docs Support

validator

validator String validation and sanitization 10.8.0

Validator Client-side Javascript Validator library. Ports from Laravel 5.2 1.0.5

validator-core Fast, Lightweight, Flexible Validator 1.13.3

validator-nu HTML5 validator using validator.nu, but not remotely 2.2.2

validator-codec Codec for validator presets 1.0.0

validator-json

validator-laravel

validator-as-promised

validator-nu-angular

validator-factory

Sign up

log in or sign up

validator.js

npm v10.8.0 build passing downloads 6M/m

A library of string validators and sanitizers.

Strings only

This library validates and sanitizes strings only.

If you're not sure if your input is a string, coerce it using `input + ''`. Passing anything other than a string is an error.

Installation and Usage

Server-side usage

Install the library with `npm install validator`

No ES6

```
var validator = require('validator');

validator.isEmail('foo@bar.com'); //=> true
```

ES6

```
import validator from 'validator';
```

install

```
> npm i validator
```

weekly downloads

827,075 

version	license
10.8.0	MIT

open issues	pull requests
31	2

homepage	repository
github.com	 github

last publish

a month ago

collaborators



Test with RunKit

Report a vulnerability

```
$ 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;
```

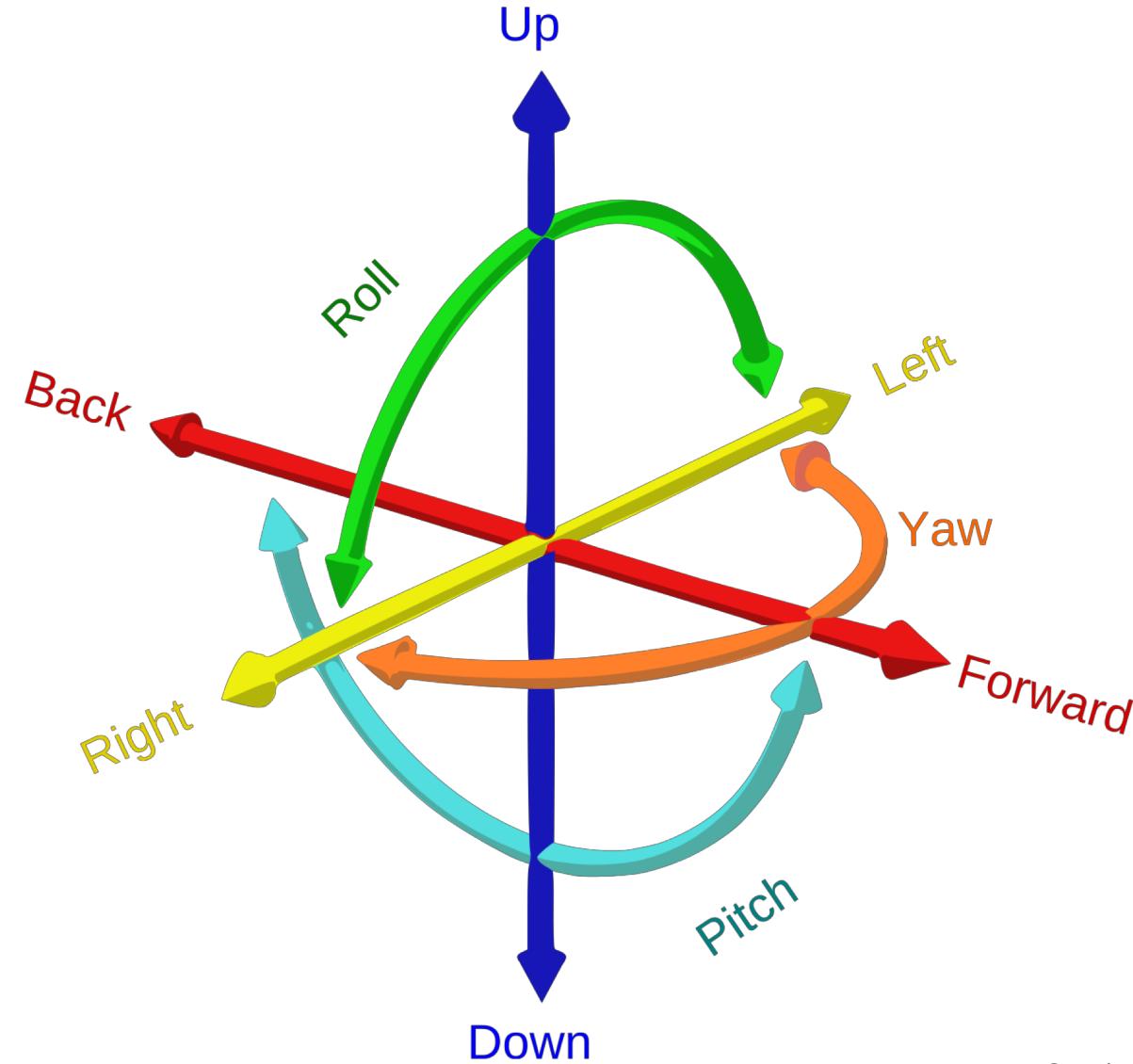
This is our tenth and final way to use GraalVM

- 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 Hummer
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.