GraalVM

GW Tech Talks

Outline

Introduction to GraalVM

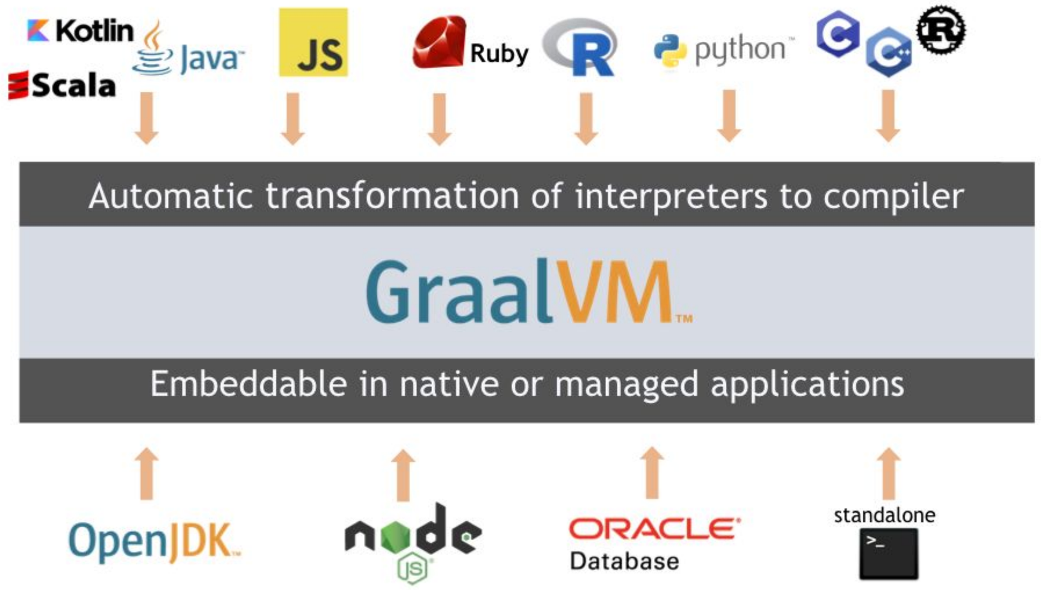
Top 10 Things To Do With GraalVM

Demo

Conclusion

GraalVM

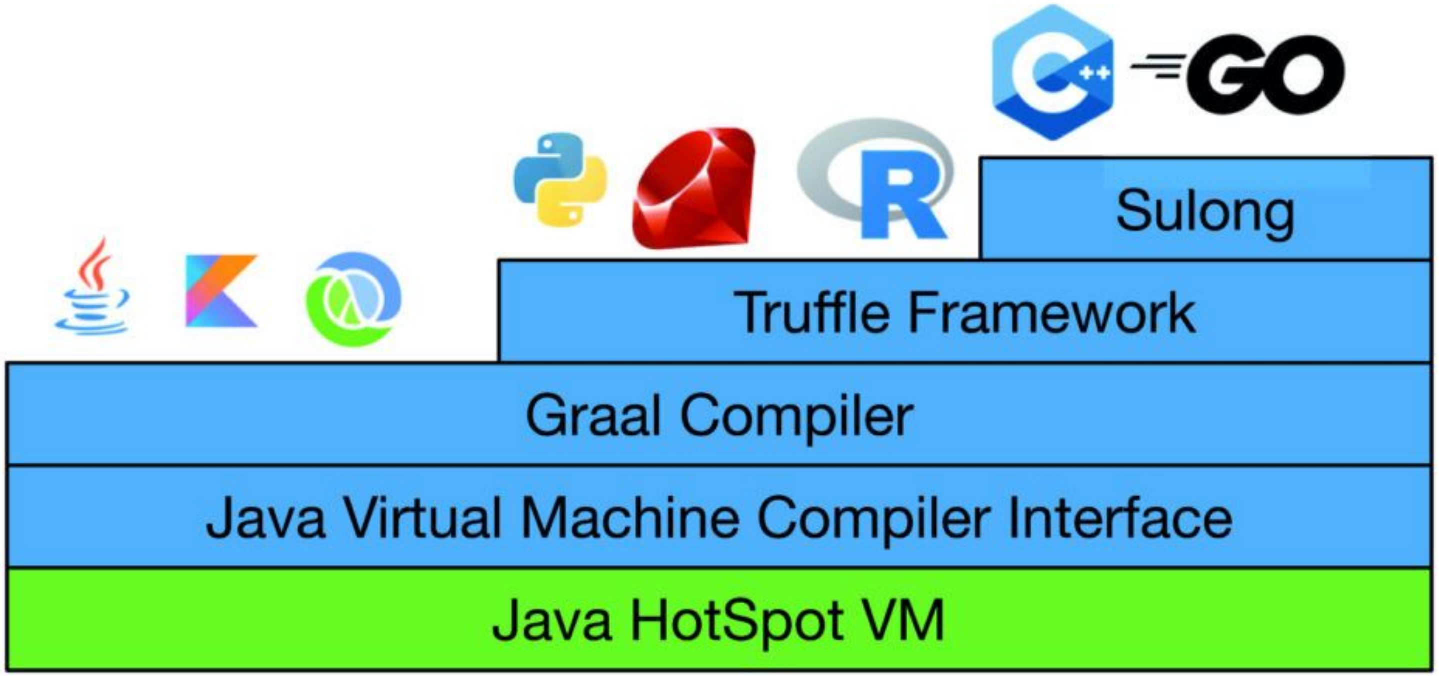
GraalVM is an ecosystem and shared runtime offering performance advantages not only to JVM-based languages such as Java, Scala, and Kotlin, but also to other programming languages such as JavaScript, Ruby, Python, and R. Additionally, it enables the execution of native code via an LLVM front-end, and WebAssembly programs on the JVM.


Kotlin
Scala
Java
JS
Ruby
R
python
Automatic transformation of interpreters to compiler
GraalVM
Embeddable in native or managed applications
OpenJDK
node
ORACLE Database
standalone

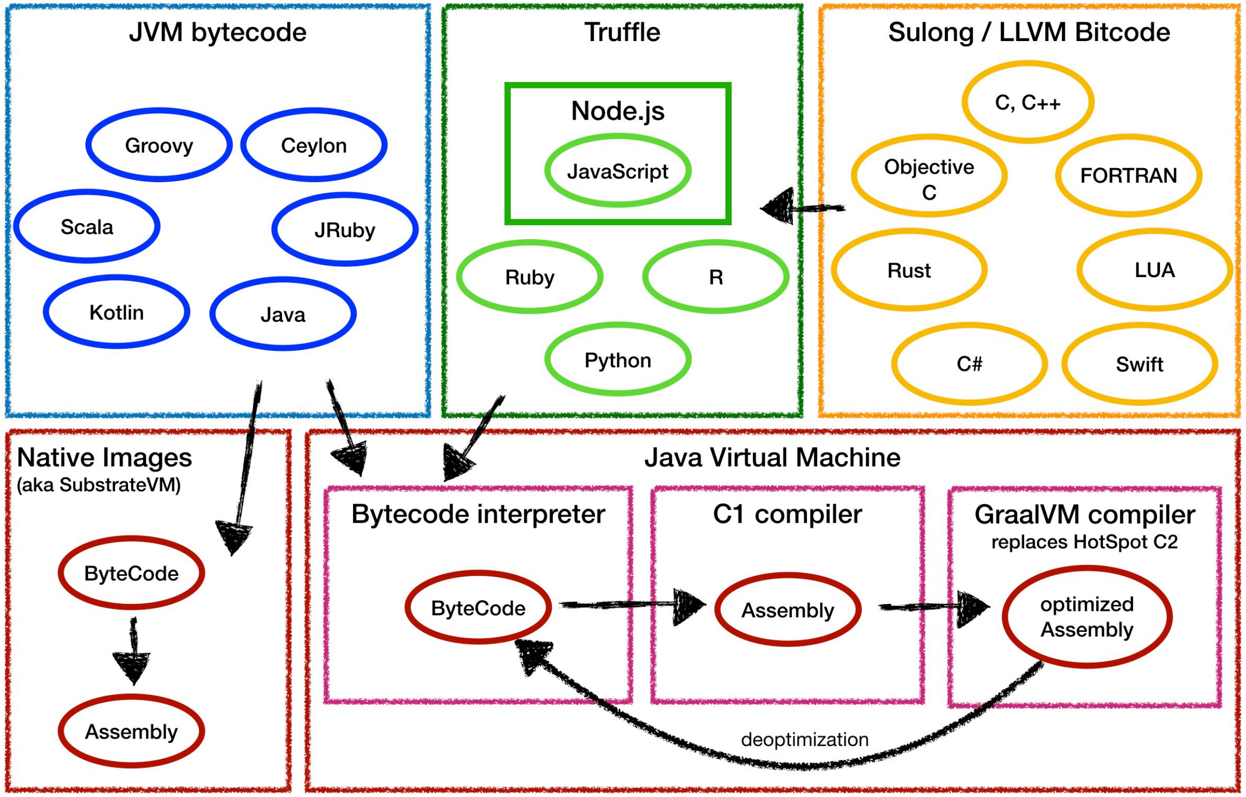
Why GraalVM?

- Run your code faster and more efficiently
- Interoperate directly with most modern programming languages
- Embed languages with the GraalVM SDK
- Create compiled native images
- Use a single set of tools to monitor, debug, and profile all your code

GraalVM Architecture



GraalVM Architecture



Running Applications

GraalVM’s /bin directory is similar to that of a standard JDK, but includes a set of additional launchers:

- **js** runs a JavaScript console with GraalVM.
- **node** is a drop-in replacement for Node.js, using GraalVM’s JavaScript engine.
- **lli** is a high-performance LLVM bitcode interpreter integrated with GraalVM.
- **gu** (GraalVM Updater) can be used to install language packs for Python, R, and Ruby.

Combine Languages

If enabled, using the **--polyglot** flag, scripts executed on GraalVM can use interoperability features to call into other languages and exchange data with them.

For example,

```
js --jvm --polyglot example.js
```

Native Images

GraalVM can compile Java bytecode into native images to achieve faster startup and smaller footprint for your applications.

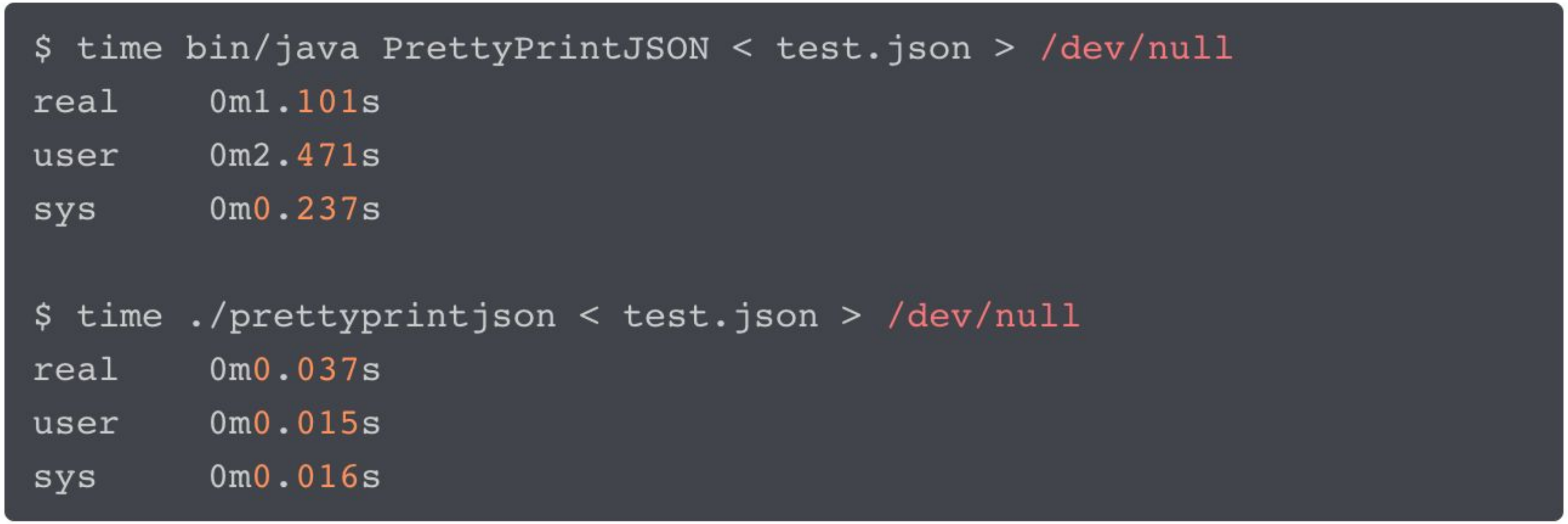
```java
// HelloWorld.java
public class HelloWorld {
  public static void main(String[] args) {
    System.out.println("Hello, World!");
  }
}
```

```
$ javac HelloWorld.java
$ native-image HelloWorld
```

```
$ ./helloworld
Hello, World!
```

Native Images

The native image is much faster than running the same code on the JVM directly:



```
$ time bin/java PrettyPrintJSON < test.json > /dev/null
real    0m1.101s
user    0m2.471s
sys     0m0.237s

$ time ./prettyprintjson < test.json > /dev/null
real    0m0.037s
user    0m0.015s
sys     0m0.016s
```

Top 10 Things To Do With GraalVM

1. **High-performance modern Java**
2. **Low-footprint, fast-startup Java**
3. **Combine JavaScript, Java, Ruby, and R**
4. **Run native languages on the JVM**
5. **Tools that work across all languages**
6. **Extend a JVM-based application**
7. Extend a native application
8. Java code as a native library
9. Polyglot in the database
10. Create your own language

High-performance modern Java

Run with Graal JIT compiler

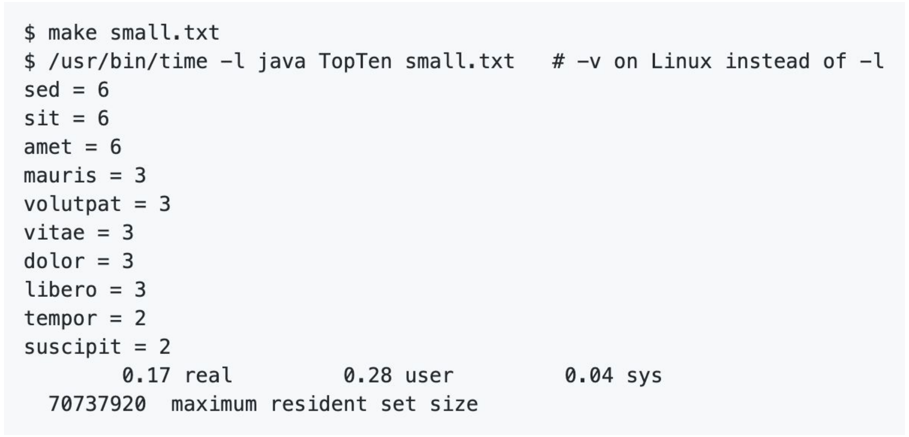
```
$ make large.txt
$ time java TopTen large.txt
sed = 502701
ut = 392657
in = 377651
et = 352641
id = 317627
eu = 317627
eget = 302621
vel = 300120
a = 287615
sit = 282613

real  0m12.950s
user  0m17.827s
sys 0m0.622s
```

Run without Graal JIT compiler
(-XX:-UseJVMCICompiler)

```
$ time java -XX:-UseJVMCICompiler TopTen large.txt
sed = 502701
ut = 392657
in = 377651
et = 352641
id = 317627
eu = 317627
eget = 302621
vel = 300120
a = 287615
sit = 282613

real  0m19.602s
user  0m20.357s
sys 0m0.498s
```

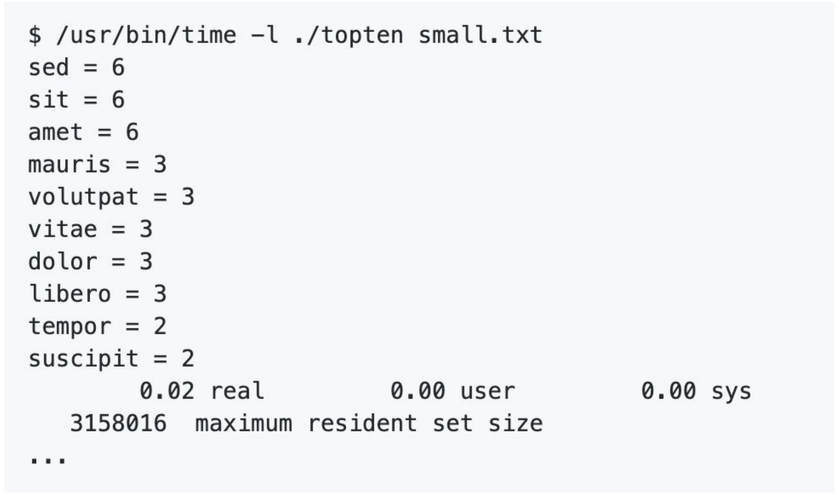
Low-footprint, fast-startup Java

Compile just-in-time



```
$ make small.txt
$ /usr/bin/time -l java TopTen small.txt   # -v on Linux instead of -l
sed = 6
sit = 6
amet = 6
mauris = 3
volutpat = 3
vitae = 3
dolor = 3
libero = 3
tempor = 2
suscipit = 2
        0.17 real         0.28 user         0.04 sys
  70737920  maximum resident set size
```

Compile ahead-of-time

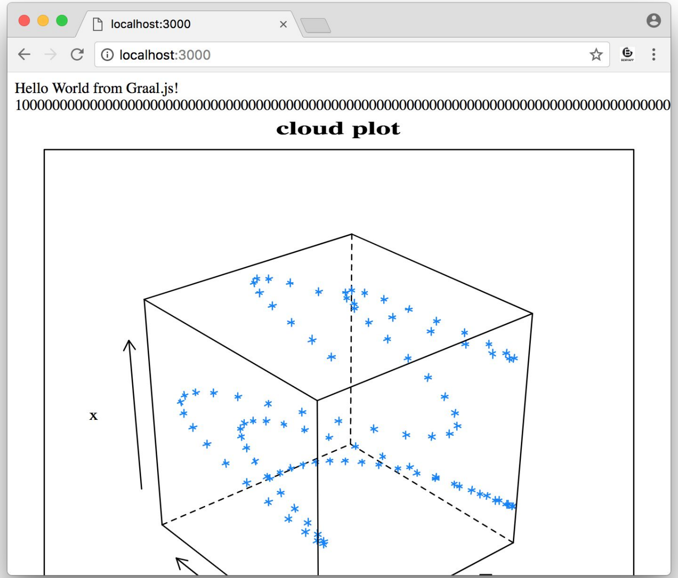


```
$ /usr/bin/time -l ./topten small.txt
sed = 6
sit = 6
amet = 6
mauris = 3
volutpat = 3
vitae = 3
dolor = 3
libero = 3
tempor = 2
suscipit = 2
        0.02 real         0.00 user         0.00 sys
   3158016  maximum resident set size
...
```

Combine JavaScript, Java, Ruby, and R

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

const BigInteger = Java.type('java.math.BigInteger')

app.get('/', function (req, res) {
  var text = 'Hello World from Graal.js!<br> '

  // Using Java standard library classes
  text += BigInteger.valueOf(10).pow(100)
          .add(BigInteger.valueOf(43)).toString() + '<br>'

  // Using R interoperability to create graphs
  text += Polyglot.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()
    `);

  res.send(text)
})

app.listen(3000, function () {
  console.log('Example app listening on port 3000!')
})
```

```
$ node --jvm --polyglot polyglot.js
```


localhost:3000

Hello World from Graal.js!
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000

Run native languages on the JVM

GraalVM can run C code in the same way that it runs languages like JavaScript and Ruby.

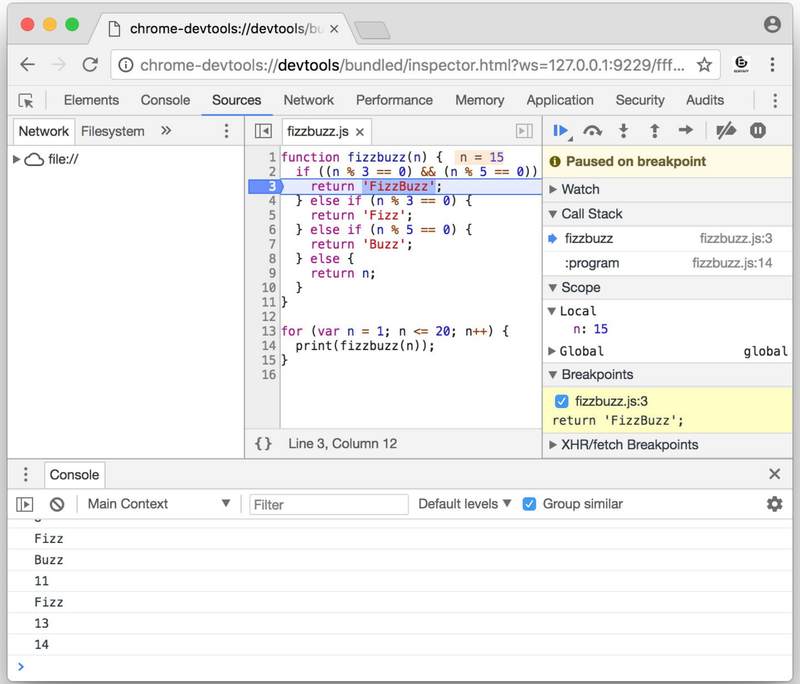
Compile using standard clang (the LLVM C compiler)

```
$ clang -c -emit-llvm gzip.c
```

Run executable using GraalVM using the lli command (LLVM bitcode interpreter)

```
$ cat small.txt
Lorem ipsum dolor sit amet...
$ gzip small.txt
$ lli gzip.bc -d small.txt.gz
$ cat small.txt
Lorem ipsum dolor sit amet...
```

Tools that work across all languages - Chrome debugger



```
function fizzbuzz(n) {  n = 15
  if ((n % 3 == 0) && (n % 5 == 0))
    return 'FizzBuzz';
  } else if (n % 3 == 0) {
    return 'Fizz';
  } else if (n % 5 == 0) {
    return 'Buzz';
  } else {
    return n;
  }
}

for (var n = 1; n <= 20; n++) {
  print(fizzbuzz(n));
}
```

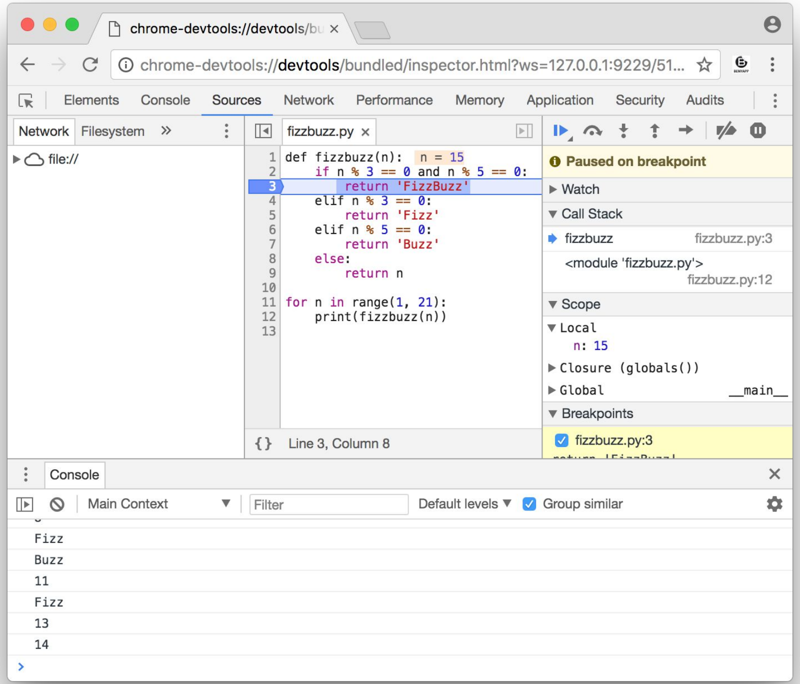
Line 3, Column 12

Paused on breakpoint

- Watch
- Call Stack
  - fizzbuzz fizzbuzz.js:3
  - :program fizzbuzz.js:14
- Scope
  - Local
    - n: 15
  - Global global
- Breakpoints
  - ✓ fizzbuzz.js:3 `return 'FizzBuzz';`
- XHR/fetch Breakpoints

Console — Main Context — Filter — Default levels — ✓ Group similar

```
Fizz
Buzz
11
Fizz
13
14
>
```



```
def fizzbuzz(n):  n = 15
    if n % 3 == 0 and n % 5 == 0:
        return 'FizzBuzz'
    elif n % 3 == 0:
        return 'Fizz'
    elif n % 5 == 0:
        return 'Buzz'
    else:
        return n

for n in range(1, 21):
    print(fizzbuzz(n))
```

Line 3, Column 8

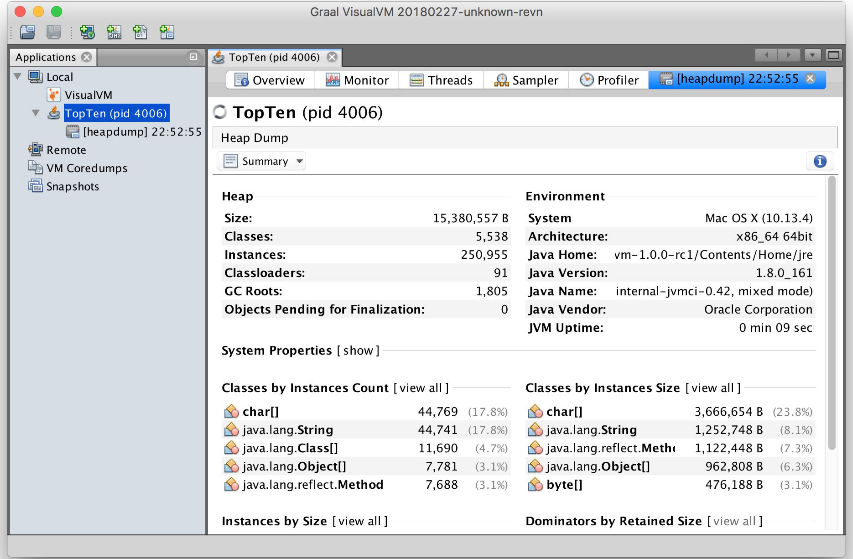
Paused on breakpoint

- Watch
- Call Stack
  - fizzbuzz fizzbuzz.py:3
  - <module 'fizzbuzz.py'> fizzbuzz.py:12
- Scope
  - Local
    - n: 15
  - Closure (globals())
  - Global __main__
- Breakpoints
  - ✓ fizzbuzz.py:3

Console — Main Context — Filter — Default levels — ✓ Group similar

```
Fizz
Buzz
11
Fizz
13
14
>
```

Tools that work across all languages - VisualVM


Graal VisualVM 20180227-unknown-revn

Applications | TopTen (pid 4006)

- Local
  - VisualVM
  - TopTen (pid 4006)
    - [heapdump] 22:52:55
- Remote
- VM Coredumps
- Snapshots

Overview | Monitor | Threads | Sampler | Profiler | [heapdump] 22:52:55

TopTen (pid 4006)

Heap Dump

Summary

Heap

Size:	15,380,557 B
Classes:	5,538
Instances:	250,955
Classloaders:	91
GC Roots:	1,805
Objects Pending for Finalization:	0

Environment

System	Mac OS X (10.13.4)
Architecture:	x86_64 64bit
Java Home:	vm-1.0.0-rc1/Contents/Home/jre
Java Version:	1.8.0_161
Java Name:	internal-jvmci-0.42, mixed mode)
Java Vendor:	Oracle Corporation
JVM Uptime:	0 min 09 sec

System Properties [show]

Classes by Instances Count [view all]

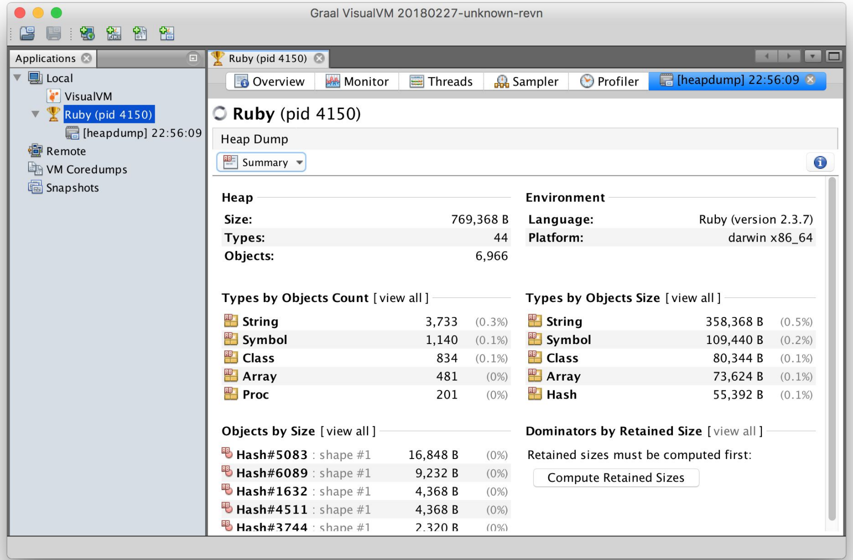
char[]	44,769	(17.8%)
java.lang.**String**	44,741	(17.8%)
java.lang.**Class[]**	11,690	(4.7%)
java.lang.**Object[]**	7,781	(3.1%)
java.lang.reflect.**Method**	7,688	(3.1%)

Classes by Instances Size [view all]

char[]	3,666,654 B	(23.8%)
java.lang.**String**	1,252,748 B	(8.1%)
java.lang.reflect.**Meth**	1,122,448 B	(7.3%)
java.lang.**Object[]**	962,808 B	(6.3%)
byte[]	476,188 B	(3.1%)

Instances by Size [view all]

Dominators by Retained Size [view all]


Graal VisualVM 20180227-unknown-revn

Applications | Ruby (pid 4150)

- Local
  - VisualVM
  - Ruby (pid 4150)
    - [heapdump] 22:56:09
- Remote
- VM Coredumps
- Snapshots

Overview | Monitor | Threads | Sampler | Profiler | [heapdump] 22:56:09

Ruby (pid 4150)

Heap Dump

Summary

Heap

Size:	769,368 B
Types:	44
Objects:	6,966

Environment

Language:	Ruby (version 2.3.7)
Platform:	darwin x86_64

Types by Objects Count [view all]

String	3,733	(0.3%)
Symbol	1,140	(0.1%)
Class	834	(0.1%)
Array	481	(0%)
Proc	201	(0%)

Types by Objects Size [view all]

String	358,368 B	(0.5%)
Symbol	109,440 B	(0.2%)
Class	80,344 B	(0.1%)
Array	73,624 B	(0.1%)
Hash	55,392 B	(0.1%)

Objects by Size [view all]

Hash#5083 : shape #1	16,848 B	(0%)
Hash#6089 : shape #1	9,232 B	(0%)
Hash#1632 : shape #1	4,368 B	(0%)
Hash#4511 : shape #1	4,368 B	(0%)
Hash#3744 : shape #1	2,320 B	(0%)

Dominators by Retained Size [view all]

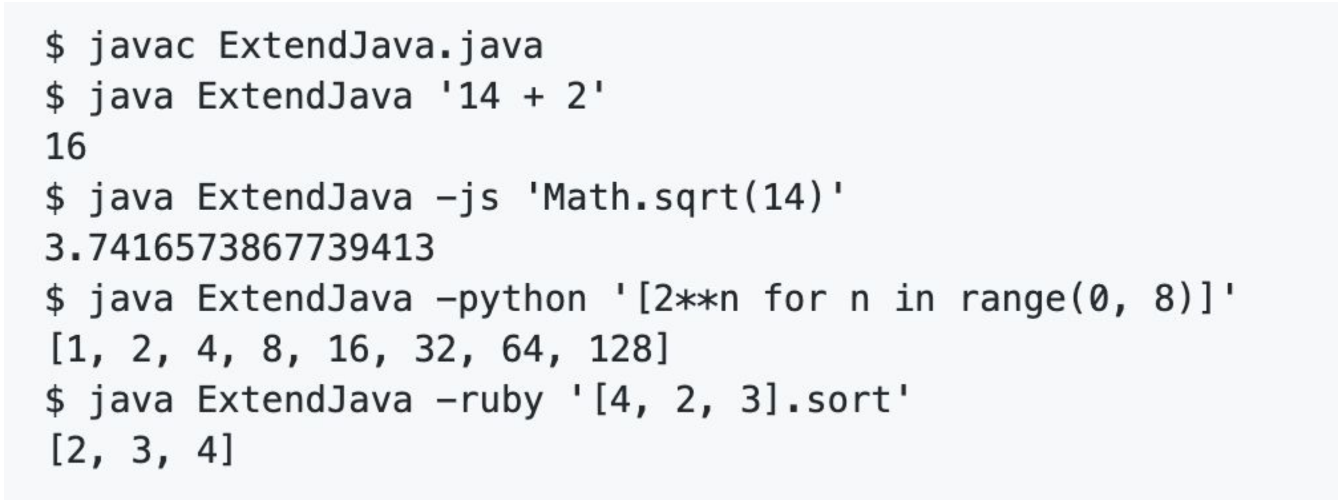
Retained sizes must be computed first:

Compute Retained Sizes

Extend a JVM-based application

```java
import org.graalvm.polyglot.Context;
import org.graalvm.polyglot.Value;

public class ExtendJava {
    public static void main(String[] args) {
        String language = "js";
        try (Context context = Context.newBuilder().allowNativeAccess(true).build()) {
            for (String arg : args) {
                if (arg.startsWith("-")) {
                    language = arg.substring(1);
                } else {
                    Value v = context.eval(language, arg);
                    System.out.println(v);
                }
            }
        }
    }
}
```

Extend a JVM-based application



```
$ javac ExtendJava.java
$ java ExtendJava '14 + 2'
16
$ java ExtendJava -js 'Math.sqrt(14)'
3.7416573867739413
$ java ExtendJava -python '[2**n for n in range(0, 8)]'
[1, 2, 4, 8, 16, 32, 64, 128]
$ java ExtendJava -ruby '[4, 2, 3].sort'
[2, 3, 4]
```

Demo

Conclusion

GraalVM provides us

- Run Java code faster and more efficiently
  - High-performance modern Java
  - Low-footprint, fast-startup Java
- High-performance polyglot VM
  - Combine Java, JavaScript, Python and R
  - Extend a JVM-based application (Java + JS)
- A single set of tools to monitor, debug, and profile Java, Python, JS code
  - VisualVM