Multilanguage Frameworks

New Ways
New Possibilities

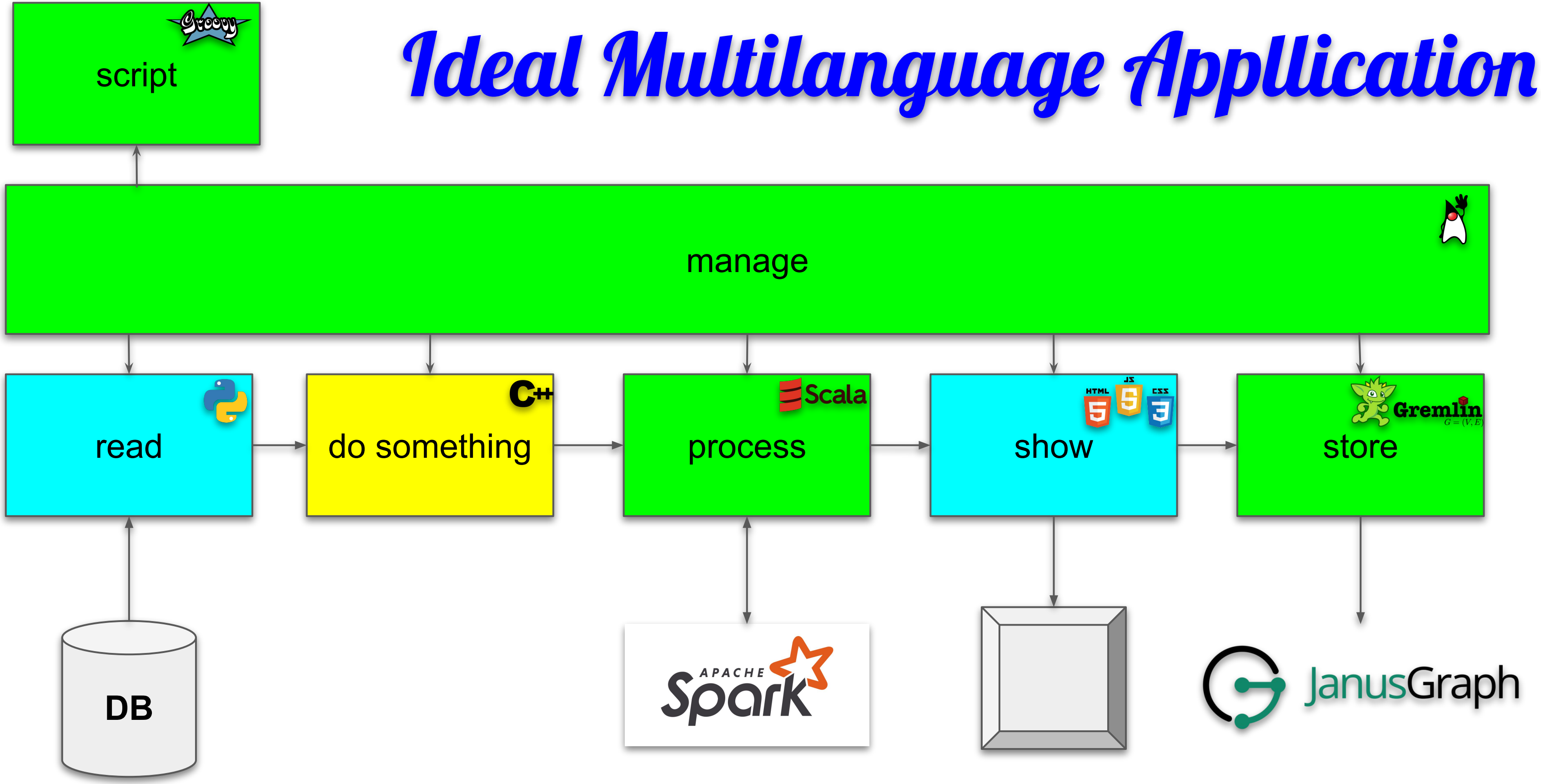
Julius Hrivnac (IJClab Orsay)

- Ideal Multilanguage Application
- JVM Multilanguage Environment
  - JVM Languages
  - Managed Languages
  - C-World
- GraalVM
- Plurality World
  - Where it is already useful
  - Intrinsic limitations
  - External complications
- Future of programming

IJCLab Irène Joliot-Curie Laboratoire de Physique des 2 Infinis

cnrs

Ideal Multilanguage Appllication



Use the best tools and languages for each task.
Transparent interfaces (no stubs).
Data sharing (no proxies).
***It works !** - We are (almost) there !*

What is
the general multilanguage technology status
?

JVM Languages

- Languages completely interoperable with Java (loaded into the same runtime or compiled into standard class-files)
- Fully inter-operable
- We can freely mix code from all those languages (even via inheritance)
- Can be used in a scripting interpreted way or compiled
- Successful new features from those languages are being incorporated in Java itself (e.g. functional syntax from Scala)

- **Groovy** (Apache): very high level scripting language, used in Graph DB
- **Scala** (Apache): functional language, used in Spark
- **Kotlin** (Google): for Android
- **Clojure**: Lisp-like
- **BeanShell**: interpreted/scripted Java

```
#!/usr/bin/env groovy
// converting SQL into XML with Groovy
// either run as a shell script or compiled
// ---------------------------------------------------------
sql = Sql.newInstance("jdbc:mysql://localhost/Tuples",
                      "org.gjt.mm.mysql.Driver")
xml = new MarkupBuilder(new File("Tuples.xml"))
xml.tagSet() {
  sql.eachRow("select * from tuple where run > 2") {
    row -> xml.tag(Run:row.run, Event:row.event)
    }
  }
```

Managed Languages

- ➢ Languages from different origin, made interoperable by re-implementation (or via specific bridges)
  - Go, Haskel, JavaScript, Lisp, OCaml, Pascal, PHP, Python, R, Rexx, Ruby, Scheme, Smalltalk, Tcl,...
- ➢ More than 100 languages available in some way

C-World

- ➢ Direct compilation to native code
  - Sometimes by pre-compiling to C
- ➢ Lack of high level management (reflection, introspection)
  - Often implemented on top with in-house solutions
    - Which generates incompatibilities
- ➢ Often considered as faster and smaller
  - But even when it’s true, there is a cost
    - Lack of functionality
    - Non-reproducibility
    - Non-portability
    - Very complex implementation of higher-level concepts
- ➢ Can be only connected via direct JNI or JNA
  - As they are running in an **unmanaged environment**
- ➢ Co-existence between managed JVM languages and low-level C-languages is difficult, proprietary or too primitive
  - No generic approach (so far)

Revolution ?
(Holy Grail ?)

GraalVM

GraalVM™

New Managed Environment
Supporting both JVM and C-based languages
To run in VM or natively

- **Universal VM**
  - Non-JVM languages are at the same level as JVM languages (=> full interoperability)
  - All languages running in the same VM (traditional multi-language environment runs multiple languages side-by-side with frequent conversions of data)
  - GraalVM is faster and smaller than OpenJVM (GraalVM is written in Java, OpenJVM is written in C++)
  - Full interoperability between OpenJVM and GraalVM (program compiled for one can be run in another)
  - Can be embedded in external applications (Oracle, Apache, MySQL,...)
- **Can build native executables and libraries** (using AOT (Ahead Of Time) compiler instead of JIT)
  - Fully interoperable with native applications
  - Smaller footprint, faster startup, sometimes faster execution
  - Losing some dynamical features

GraalVM

GraalVM™

- Polyglot (J)DK & (J)VM
- By Oracle
  - Big effort
  - Also included in OracleDB
  - Already used in industry (Twitter,...)
- CE (Community Edition): GPL licence (or less) - as Java
  - Components have the same licences as the original implementations (eg. Python as Python)
- EE (Enterprise Edition): better performance, security, support,...
- GraalVM JIT is included in OpenJDK (project *Galahad*):

```
java -XX:+UnlockExperimentalVMOptions -XX:+UseJVMCICompiler
```

  - So trivial to try
  - Native Image compiler will follow
- New release every 3 months
  - rel22 supporting JDK 11,17
  - rel23-dev supporting JDK 17,20
- Linux, MS, MacOSX
- Uses new Java modularity features (since Java 9)
  - As the pluggable JIT compiler
- Similar project in the past: NestedVM - failed in 2009

Supported Languages

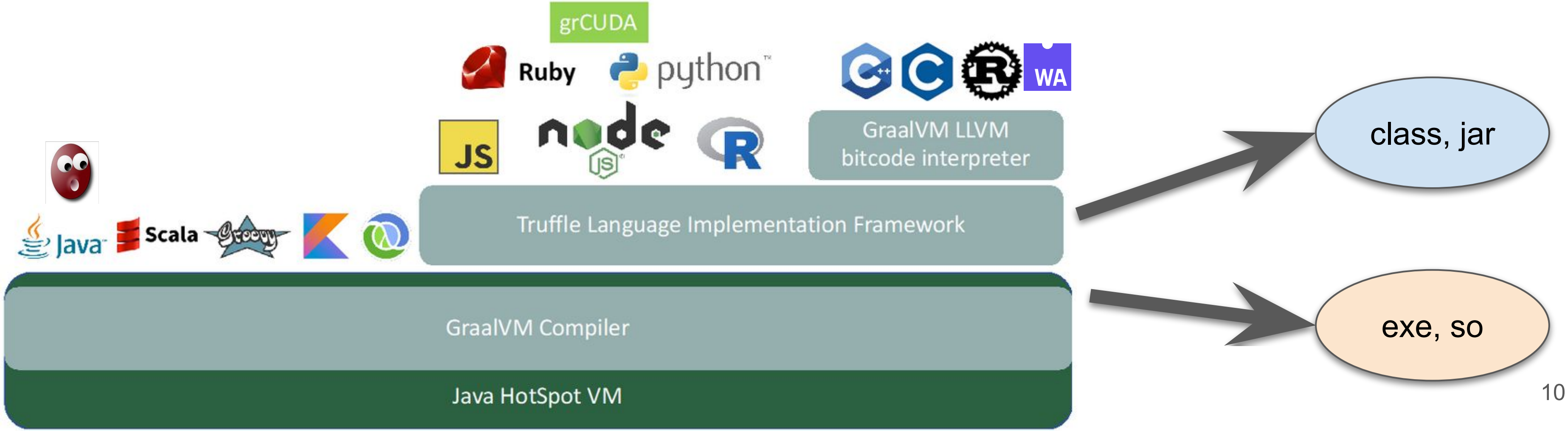
GraalVM™

- ➢ Growing number of supported languages (CUDA, WebAssembly,...)
- ➢ New Tools (debuggers, profilers, monitors,...)
- ➢ Integration in other applications and toolkits

Multiple languages are running in the same space/environment.

X

Traditional multi-language pgms run multiple languages side-by-side.



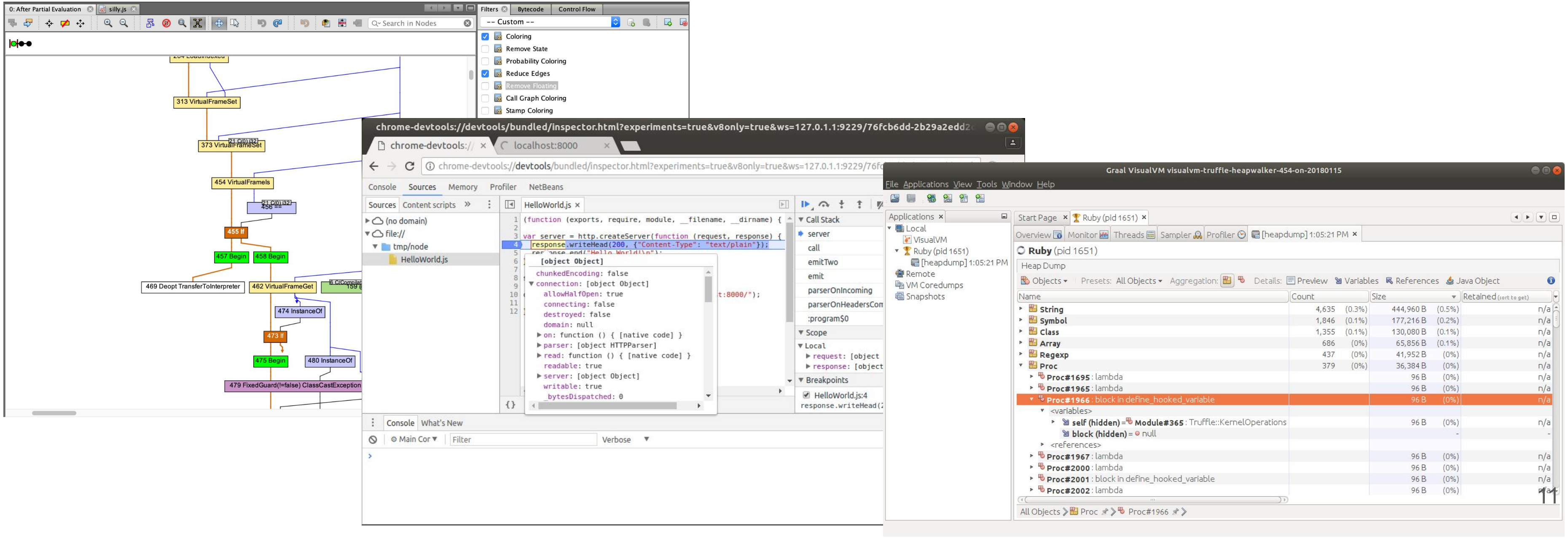
Tools

GraalVM™

- ➢ Growing number of supported languages (CUDA, WebAssembly,...)
- ➢ New Tools (debuggers, profilers, monitors,...)
- ➢ Integration in other applications and toolkits

Tools understand your language.

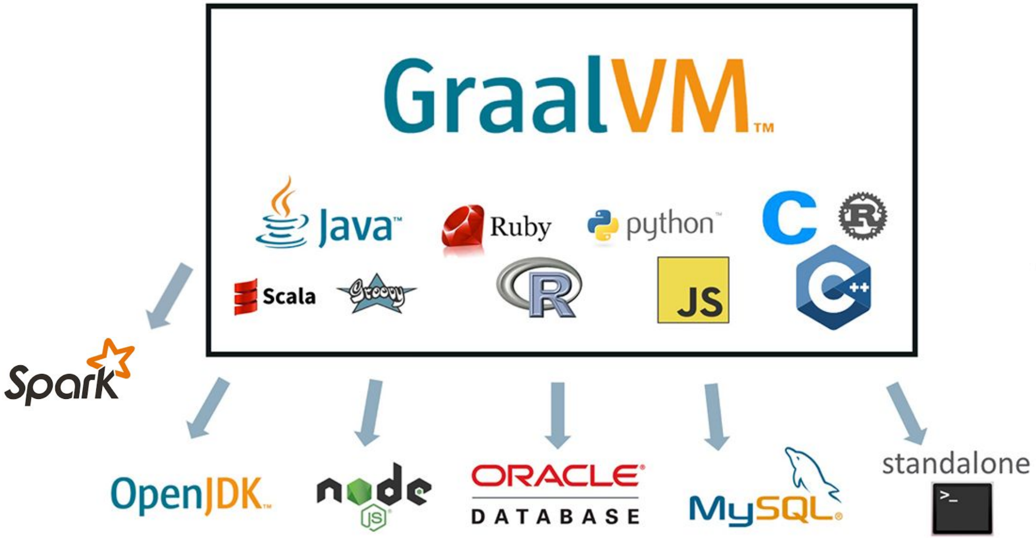
Unlike tools for pre-compiled languages.



Integration

- Growing number of supported languages (CUDA, WebAssembly,...)
- New Tools (debuggers, profilers, monitors,...)
- Integration in other applications and toolkits

Allows, for example,
using MySQL with Python instead of SQL.



GraalVM™

Native Image Example

Basic Example

```
$ javac Hello.java
$ time java Hello
Hello !
0,10s user 0,03s system 131% cpu 0,097 total
$ native-image Hello
===============================================================================
GraalVM Native Image: Generating 'hello'...
===============================================================================
[1/7] Initializing...                                              (4.1s @ 0.21GB)
 Version info: 'GraalVM 22.0.0.2 Java 11 CE'
[2/7] Performing analysis...  [*******]                           (12.7s @ 0.47GB)
   2,563 (82.60%) of  3,103 classes reachable
   3,211 (60.36%) of  5,320 fields reachable
  11,648 (72.43%) of 16,082 methods reachable
      27 classes,    0 fields, and   135 methods registered for reflection
      57 classes,   58 fields, and    51 methods registered for JNI access
[3/7] Building universe...                                         (0.8s @ 0.62GB)
[4/7] Parsing methods...      [*]                                  (0.8s @ 0.84GB)
[5/7] Inlining methods...     [****]                               (1.2s @ 0.75GB)
[6/7] Compiling methods...    [***]                                (9.3s @ 1.19GB)
[7/7] Creating image...                                            (1.1s @ 1.45GB)
   3.69MB (35.06%) for code area:    6,949 compilation units
   5.86MB (55.66%) for image heap:   1,543 classes and 80,509 objects
 999.26KB ( 9.28%) for other data
  10.52MB in total
-------------------------------------------------------------------------------
Top 10 packages in code area:                  Top 10 object types in image heap:
 606.25KB java.util                               1.64MB byte[] for general heap data
 282.31KB java.lang                             715.44KB java.lang.String
 222.52KB java.util.regex                       548.99KB java.lang.Class
 219.55KB java.text                             451.55KB byte[] for java.lang.String
 193.17KB com.oracle.svm.jni                    363.75KB java.util.HashMap$Node
 149.73KB java.util.concurrent                  192.00KB java.util.HashMap$Node[]
 117.92KB java.math                             139.81KB java.lang.String[]
 103.60KB com.oracle.svm.core.reflect           139.04KB char[]
  97.83KB sun.text.normalizer                   130.59KB java.util.concurrent.ConcurrentHashMap$Node
  88.78KB com.oracle.svm.core.genscavenge       103.92KB sun.util.locale.LocaleObjectCache$CacheEntry
      ... 111 additional packages                    ... 723 additional object types
                                  (use GraalVM Dashboard to see all)
-------------------------------------------------------------------------------
                 1.6s (5.1% of total time) in 17 GCs | Peak RSS: 2.54GB | CPU load: 3.33
-------------------------------------------------------------------------------
Produced artifacts:
 hello (executable)
 hello.build_artifacts.txt
===============================================================================
Finished generating 'hello' in 31.1s.
$ time hello
Hello !
0,00s user 0,00s system 89% cpu 0,002 total
```

Real-life Example

```
${graalvm_dir}/bin/native-image \
--delay-class-initialization-to-runtime=\
io.grpc.netty.shaded.io.netty.handler.ssl.OpenSsl \
--initialize-at-build-time=\
org.apache.log4j.Level,\
org.apache.log4j.Layout,\
org.apache.log4j.PatternLayout,\
org.apache.log4j.Logger,\
org.apache.log4j.helpers.LogLoorg.apache.log4j.Level,\
org.apache.log4j.Priority,\
org.apache.log4j.LogManager,\
org.apache.log4j.helpers.Loader,\
org.apache.log4j.helpers.LogLog,\
org.apache.log4j.Category,\
org.apache.log4j.spi.RootLogger,\
org.apache.log4j.spi.LoggingEvent,\
org.slf4j.LoggerFactory,\
org.slf4j.impl.Log4jLoggerAdapter,\
org.slf4j.impl.StaticLoggerBinder,\
java.beans.Introspector,\
com.sun.beans.Introspector,\
com.sun.beans.introspect.ClassInfo \
--report-unsupported-elements-at-runtime \
-H:Name=GroovyEL.exe \
-H:Path=../bin \
-jar ../lib/GroovyEL.exe.jar
```

Polyglot Examples (1)

GraalVM™

- Objects are never copied
- Conversion (into client physical format) at the latest possible time
- All tools are available for all languages
- **Several ways of calling foreign language:**
    - **Load as a script and execute**
    - **Compile as a class and use**
    - **Generate Native Image and call**

```
// Java calling C
Context context = Context.create();
File file = new File("polyglot"); // c-pgm compiled with GraalVM
Source source = Source.newBuilder("llvm", file).build();
Value cpart = polyglot.eval(source);
cpart.execute();
```

```
// Java calling Python
Value clazz = context.eval(Source.newBuilder("python", new File("mycode.py")).build());
Value instance = clazz.newInstance(1234);
System.out.println(instance.invokeMember("pyMethod", new int[]{1, 2, 3}));
```

```
// C calling JS
poly_create_context(thd, &ctx);
poly_context_eval(thd, ctx, "js", "foo", "function() {return 42;}", &func);
poly_value_execute(thd, func, NULL, 0, &answer);
poly_value_fits_in_int32(thd, answer, &fits);
poly_value_as_int32(thd, answer, &result);
return result;
```

```
// Java calling JS
Context context = Context.create();
Value v = context.eval("js", "function() {return 42;}");
Value answer = v.execute();
return answer.asInt();
```

Polyglot Examples (2)

GraalVM™

- Interaction with LLVM languages requires more boiler-plate code
- It's simpler to compile JVM code into Native Image than to interface JVM with LLVM
- C++ calling Java is simpler than Java calling C++

```
// JS calls CUDA
const DeviceArray = Polyglot.eval('grcuda', string='DeviceArray')
const in_arr = DeviceArray('float', 1000)
const out_arr = DeviceArray('float', 1000)
// set arrays ...
const code = '__global__ void inc_kernel(...) ...'
const buildkernel = Polyglot.eval('grcuda', string='buildkernel')
const incKernel = buildkernel(code, 'inc_kernel', 'pointer, pointer, uint64')
incKernel(160, 256)(out_arr, in_arr, N)
```

```
// C++ calls Java

// C++
int main() {
  graal_isolate_t *isolate = NULL;
  graal_isolatethread_t *thread = NULL;
  graal_create_isolate(NULL, &isolate, &thread);
  printf("Result> %d\n",ceilingPowerOfTwo(thread, 14));
  }

// Java
public class MyMath {
  @CEntryPoint (name = "ceilingPowerOfTwo")
  public static int ceilingPowerOfTwo(IsolateThread thread, int x) {
    return IntMath.ceilingPowerOfTwo(x);
    }
  }
```

```
// JS calls C++

// JS
loadSource("llvm", "cpppart");
Value getSumOfArrayFn = polyglotCtx.getBindings("llvm").getMember("getSumOfArray");
int sum = getSumOfArrayFn.execute(sqrNumbers, sqrNumbers.length).asInt();

// C++
extern "C" getSumOfArray(int array[], int size) {
  int i, sum = 0;
  for (i = 0; i < size; i++) {
    sum += array[i];
    }
  return sum;
  }
```

Where it is already useful Now

- Good news: **It really works and it works well**
- For JVM languages:
  - Just using GraalVM JIT (included in OpenJVM) makes it faster (better optimisation)
  - Compiling with GraalVM compiler make better bytecode
  - Creating Native Image may improve performance
  - Allows better integration with other languages
  - For Scala:
    - GraalVM JIT is able to optimize Scala much more than OpenJVM JIT (factor > 2)
- For Python:
  - Full interoperability with JVM languages
  - Speed, especially when compiled to Native Image
  - Better interoperability with C/C++ when compiled to Native Image
- For C/C++:
  - Can replace C/C++ code with code in better languages or integrate existing components written in better languages
    - By compiling them into Native Image or connecting with Truffle multi-language environment
  - Integration in frameworks written in other languages
  - Possibility to run in *Managed Environment* (so easy debugging)
  - Sometimes performance gain just by re-building using GraalVM (without modification)

Can rewrite just one part of the system in another (more suitable) language,
And compile into native executable.

Intrinsic Limitations

- It may be complicated to configure
  - It many cases, native image generation should be configured/tuned
  - One can/should configure/tune for performance
- Some (Java) applications may need JVM even when compiled into native executable
  - When they (mis)use reflection and construct classes at run-time
    - For example log4J
  - But after all, we may consider JVM just as another native library (which it is)
- We may gain speed for small applications, not so often for large complex ones
  - Not surprising, Java is often fast for real-life applications
- By compiling into native executable, we lose flexibility and portability
- Truffle languages (Python, Ruby, JS,...) are not at the same level of inter-operability as direct JVM languages
- Co-existence of LLVM languages (C, C++, Rust) with JVM languages is not as straightforward as between two JVM languages
  - Different memory & object models
  - Values, objects, names should be converted
  - Heavy communication across LLVM-JVM border may slow down execution
  - In that case, it may be more useful to compile JVM languages into native image
  - But it's probably as far as one can go in integrating JVM & C languages

External Complications

- Language specific build systems
    - Very elaborated make files
- Language specific deployment systems
    - Silently installing dependencies
        - Pip, conda, node, …
- Specific bridges between languages
    - Often, internal implementation uses other languages
        - Python packages often contains C code, …
- Language versions
    - It’s impossible to support all language versions and dialects
        - Python 2 vs 3, …
- Complex project specific environments

Long list of projects which have already been ported/migrated/interfaced.
The most popular & least proprietary ones.

Future of Programming

- ➢ The Frameworks will consist of various components …
  - ○ Third-party black-boxes
  - ○ Written by AI
  - ○ Legacy boxes
- ➢ Sometimes, we may not even know (or care) what is the implementation language
  - ○ This already works in the classical JVM
- ➢ Languages will be used for their strong points (Scala for parallelism, JavaScript for Graphics,...)
- ➢ Seamless (plug-in) …
- ➢ It's important to really separate data from algorithms and logic (finally)

Can rewrite just one part of the system in another (more suitable) language,
And compile into native executable.

Successfully Tested on:

- ➢ http://hrivnac.web.cern.ch/hrivnac/Activities/Packages/FinkBrowser
  - https://github.com/hrivnac/FinkBrowser
- ➢ https://hrivnac.web.cern.ch/hrivnac/Activities/Packages/Lomikel
  - https://github.com/hrivnac/Lomikel
- ➢ http://hrivnac.web.cern.ch/hrivnac/Activities/Packages/Atlascope
  - https://gitlab.cern.ch/atlas-event-index/GraphDB

Next step: Try on a real-life big project.

Backup Slides

Java

- **➢** High-level programming **environment**
  - Java Language (and compiler) + Java Virtual Machine (runtime) + standard libraries
- **➢** Created 1995 by James Gosling for Sun
- **➢** Major implementations:
  - Oracle
  - OpenJVM (GPL) - the reference
- **➢** Evolves following formal *Java Community Process* via *Java Enhancement Proposal* (JEP) and *Java Specification Requests* (JSR)
  - All standard features should have the reference implementation and the conformity test suit
- **➢** Two release per year (March, September)
  - Current release: 17 (18 should be released today)
  - We are mostly using: 8, sometimes 11
  - Early access already for: 19
- **➢** Yearly Java One Conference @ San Francisco
- **➢** Almost completely backward compatible (i.e. one can compile/run old programs in new Java), except for some newly introduced keywords (like `assert`)
- **➢** Very dynamic and flexible environment
  - Introspection, Memory Management, …
- **➢** Many monitoring and profiling tools (thanks to introspection)

Java Performance

- Performance:
  - As other languages: math, graphics,... (as they are all calling the same implementation behind)
  - Faster than other languages: OO features, memory management, parallelism, dynamic optimisation
  - Slower than other languages: matrix manipulations (as no native matrices), some numerical operations (a cost for exact reproducibility), startup (as should load VM and perform initial optimisation)
  - Needs more memory (to enable reflection, memory management and allow dynamical features and runtime optimisation)
- Comparing performance is very difficult
  - Startup vs warmup vs peak
  - Throughput vs latency vs memory
  - Min vs max vs mean
  - Environment may be tuned for a specific performance requirements
  - Should compare on **real** applications, but then comparing not only language
    - Should include aux functionality (memory management, at least some reflection, often parallelism,...)

Java Object Model

- Very sophisticated mechanism for creating Objects from different sources via hierarchy of `ClassLoaders` (what '`new`' does)
- Allows constructing Objects like Lego
  - System classes
  - From JAR files
  - From Network
  - As Java Beans (Web Service)
  - Via Serialisation, object databases (e.g. reading of Root files)
  - Using Aspects (= enhancing objects at runtime)

```
ClassLoader loader = new MyClassLoader(...);
Object o = loader.loadClass("MyNamespace.MyClass").newInstance();
```

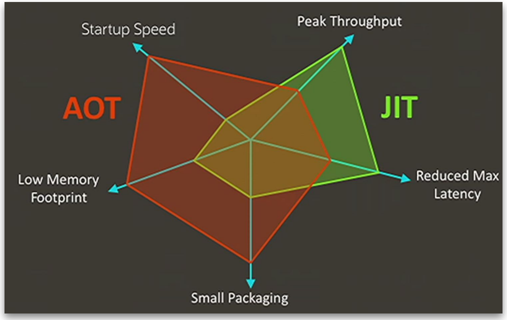
- Full class name includes classloader namespace + class name
  - So we can have different classes with the same name in one program
    - Allows for object migration (= one object changes its class)
    - Allows for dynamic re-loading of classes
- Base for reflection, memory management,...
- May be tricky and non-intuitive to use (e.g. anti-inheritance pattern)
  - Sometimes misused (log4j ?)
  - Application developer rarely needs it
- Since Java 9 extended to Java Modules (which can explicitly import/export/hide components)
- Foundation for multi-language environment
  - Classloaders loading from different languages into the same runtime

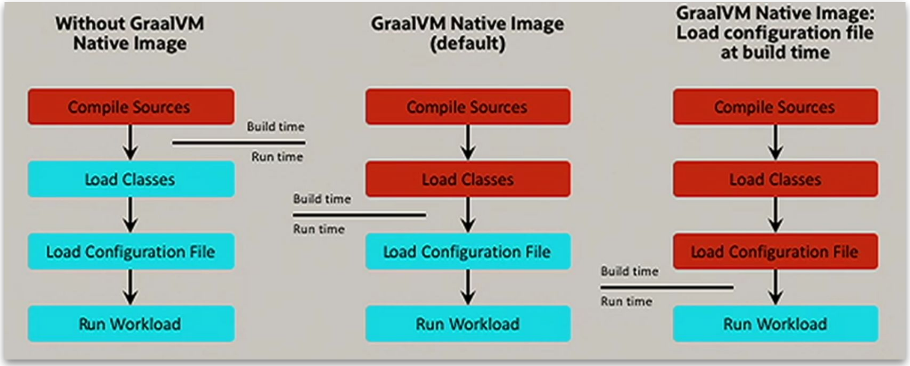
Generating GraalVM native image is better than re-writing Java/Python/... in C/C++/Go,..

JIT vs AOT

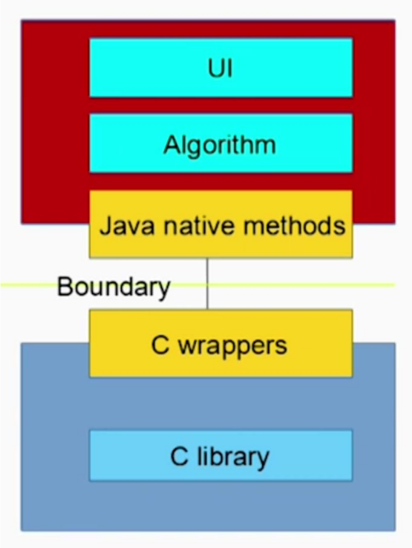
GraalVM™

- ➢ JIT = Just In Time Compiler: compiling into bytecode (jar), dynamically re-compiling at runtime by JVM (HotSpot)
- ➢ AOT = Ahead Of Time Compiler: compiling into native binary (exe, so)
  - Very complex due to extremely dynamic nature of Java - tries to guess what is going on during runtime
  - Runs initialisation and creates initial heap during compilation
  - *Close World Assumption*: All dependencies should be available at compile time (not true for JIT), no dynamic loading
  - May have to provide hints about dynamic usage (reflection operations, class initialisation, lambdas, annotations, service loaders,...)
    - Can use **Tracing Agent** for that
    - Can put this configuration in jar `META-IN/native-image`
  - Can configure to tune the image (memory vs speed,...)
  - May need JVM at runtime (*fallback image*) to handle some dynamical operations
- ➢ Can compile jar into exe, so

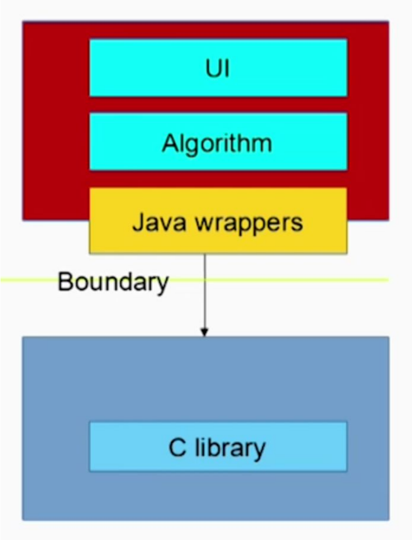




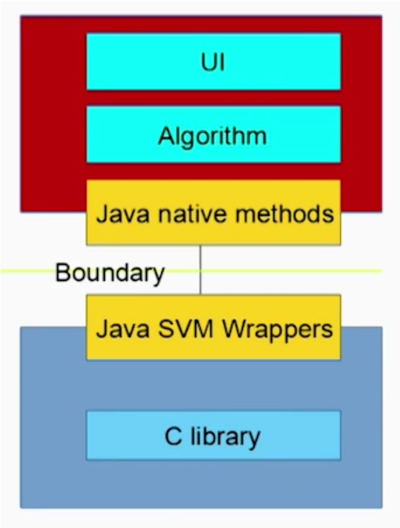
Java calling C



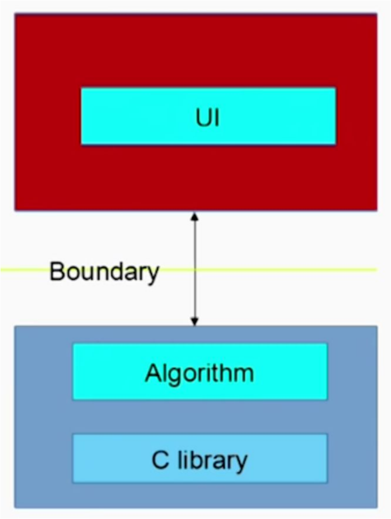
Traditional JNI
slow, complex



Traditional JNA
faster, complex



JNI via Native Image
fast, simpler



Native Image
fast, simple