GraalVM

https://github.com/azatsatklichov/Java-Features

Azat Satklichov
azats@seznam.cz,
http://sahet.net/htm/java.html

Agenda

See also, demos:
"JVM Memory Management - Garbage Collection, GC Tools, ..
"Java Performance Tuning - JMH "
"JVM Architecture, Classloading, etc. …"

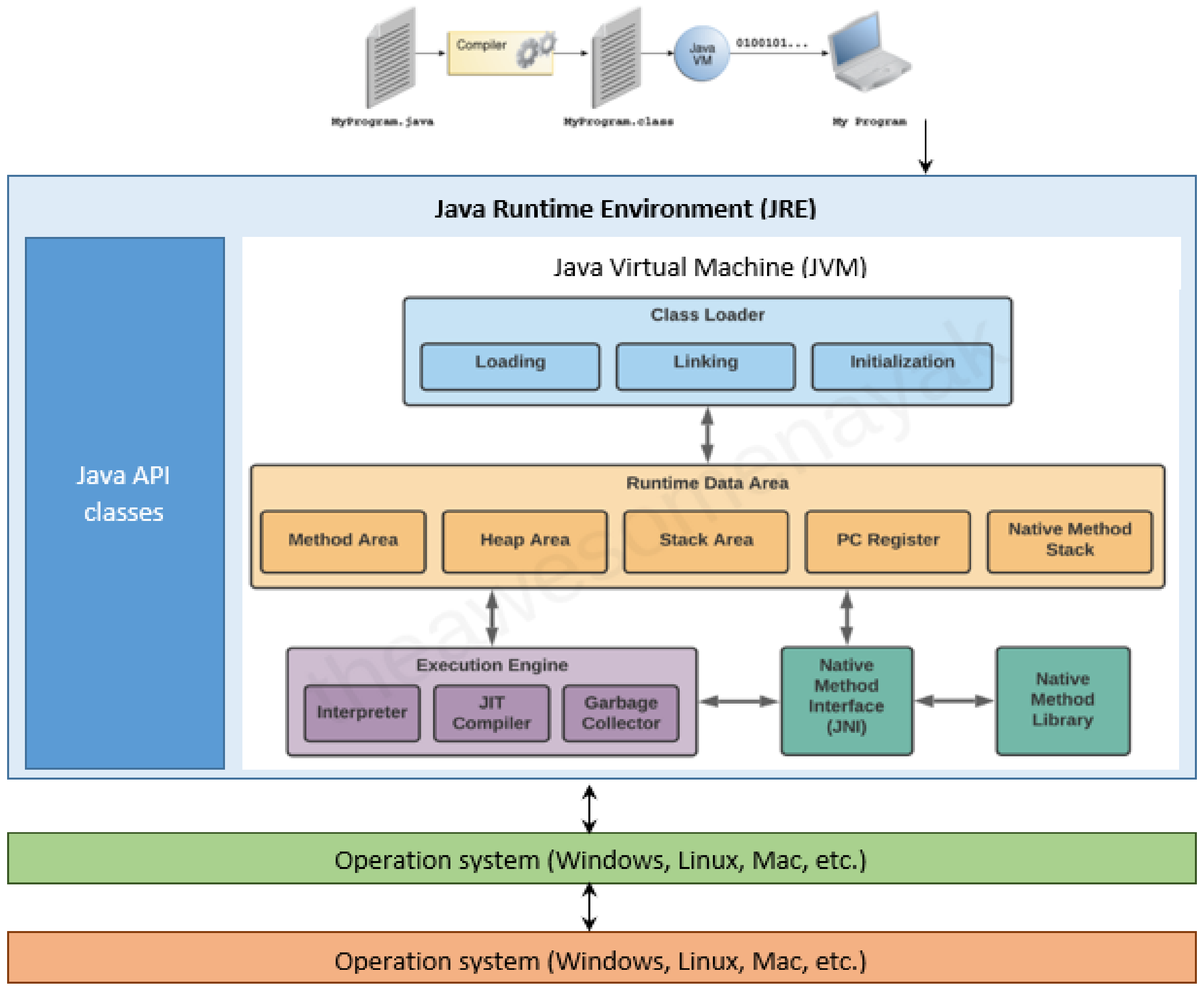
- ❑ JVM Architecture
- ❑ Execution Engine
- ❑ Interpreter and Just In Time (JIT) Compiler
- ❑ C1 and C2 Compilers
- ❑ Java 15 Features
- ❑ GraalVM Architecture
- ❑ Language and Runtime Support
- ❑ Polyglot API - Combine Languages
- ❑ Native Images
- ❑ Truffle Language Implementation Framework
- ❑ Thing To Do with GraalVM
- ❑ https://www.graalvm.org/latest/docs/quick-references/

JVM Architecture

A virtual machine is a *virtual representation of a physical computer*. A **Java virtual machine** (**JVM**) is a virtual machine that enables a computer to run Java programs as well as programs written in other languages that are also compiled to Java bytecode.

The five major components inside JVM are

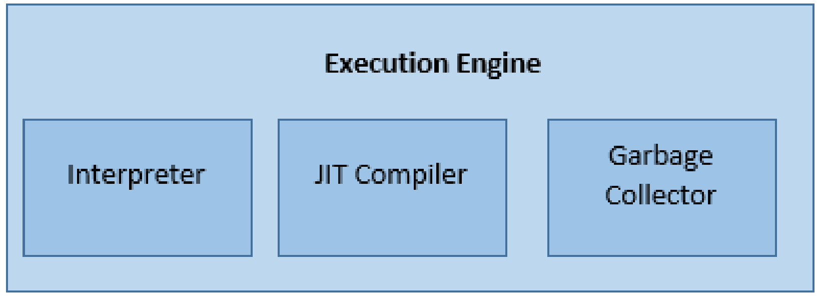
- Class Loader (loads class files to RAM)
- Memory Area (contains runtime data)
- Execution Engine (executes byte-code using interpreter)
- Native Method Interface
- Native Method Library



JVM Architecture

Execution Engine

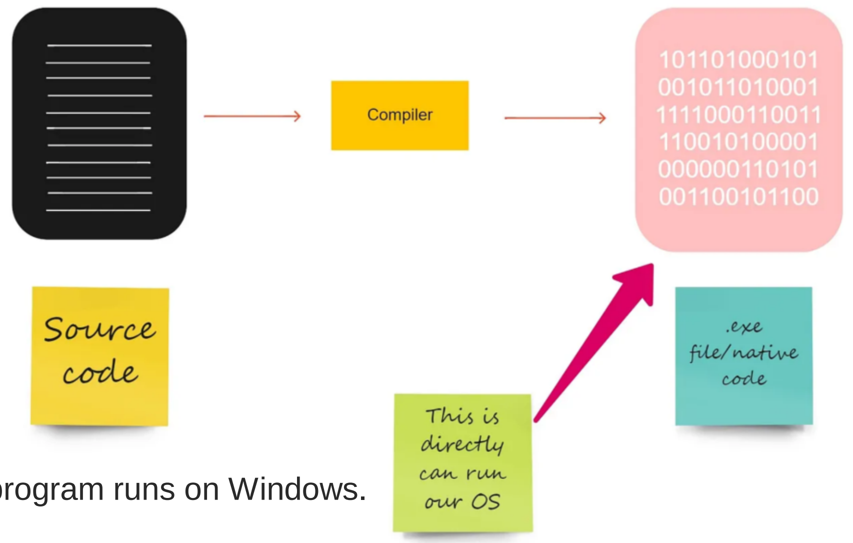
Actual execution of byte-code happens here. Execution Engine executes the instructions in the bytecode line-by-line using interpreter by reading the data assigned to above runtime data areas.
Interpreter - before executing the program, the bytecode needs to be converted into machine language instructions. The JVM can use an **interpreter or a JIT compiler** for the execution engine. It interprets the bytecode and executes the instructions one-by-one. It can **interpret one bytecode line quickly**, but executing the interpreted **result is a slower task.**



***A compiler** is a software program that translates computer code written in one programming language into another language. A compiler is used for programs that translate source code from a high-level programming language to a lower-level language.*

There are two types which we can classify how a program runs on our computer.
1) Exec. program directly on OS (run native code)   2) Exec. program on a JVM that is run on OS

How does a program run on a OS (natively)?



C program runs on Windows.

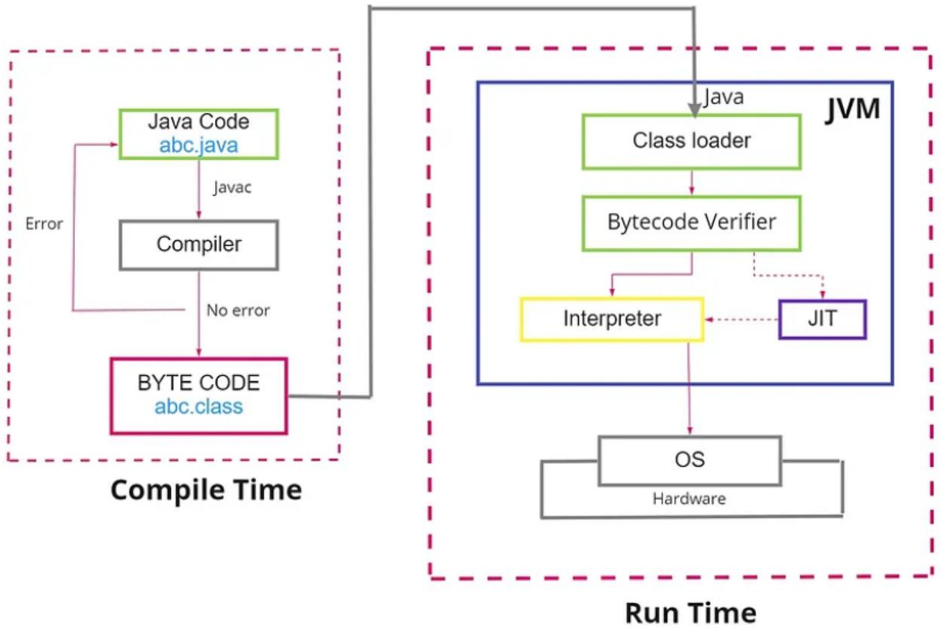
This executable file is platform dependent. Your C compiler will generate a Windows-compatible executable file if you execute it on a Windows machine. In other words, If you run your C compiler on a Linux system, it will generate executable files that are Linux compatible

The **main difference** between **compiler and interpreter** is, the compiler compiles the whole code at once into machine code(native code). While the interpreter converts code into machine code by reading line by line. So we can say that the Java program is slower than other languages like C because it uses interpreters during run time.
After the interpreter interprets the code then it transfers to the OS and is executed. So this is how the program runs in a virtual machine without directly running on an OS.

How does a program run on a Virtual Machine?

Unlike C program, Java prog. is both compiled (e.g. **abc.class**) and interpreted. Actually, this **bytecode** is not understood by our OS directly. This only can understand by JVM on RAM. According to this analogy, a virtual machine is a program that allows you to run a program (byte code) on this machine rather than directly on the operating system. So, **bytecode (class file)** is run by JVM.
This below 3-step is a **Compile Time process**.

To run it on OS directly, you need a **native-code** [at runtime compile bytecode into native] generated by JIT**.** To make the JAVA run on Windows/Linux directly, we compile (runtime compile by JIT) java byte-code [.class] into native-code which runs on OS directly. This is a **Runtime Time process.**

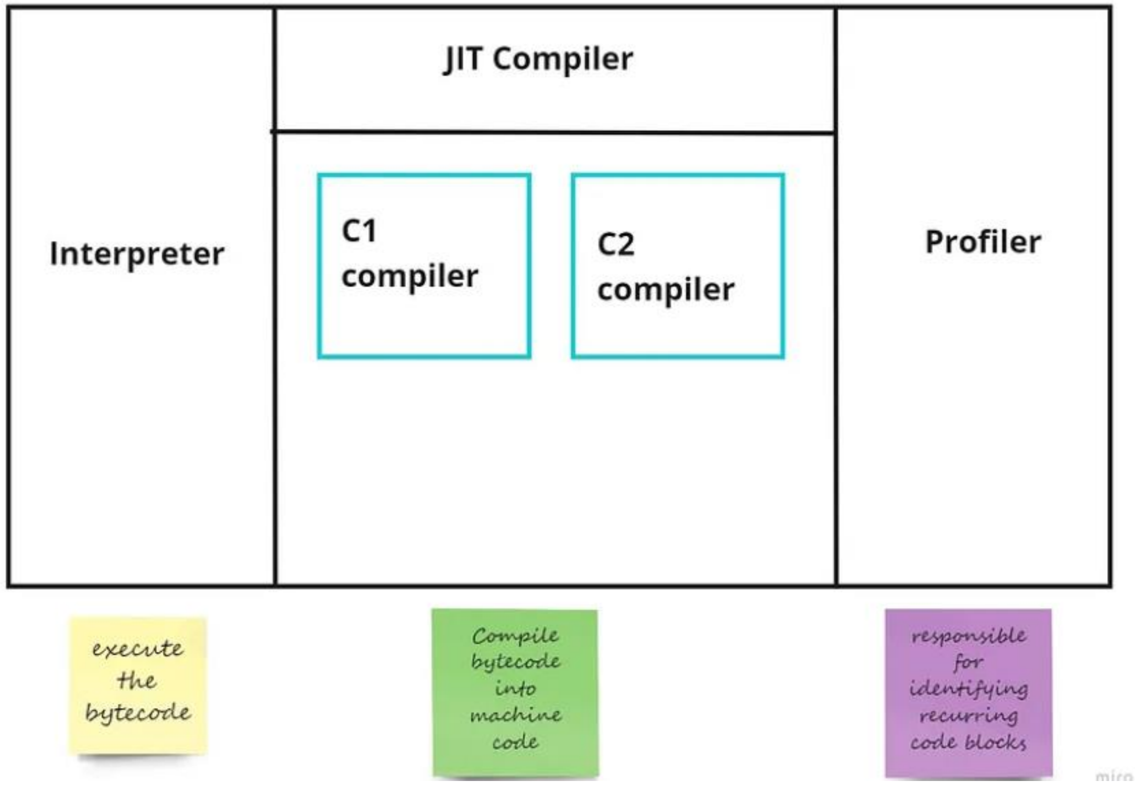


The Just in Time(JIT) compiler

JIT compiler is located inside the **Execution Engine** [Interpreter, JIT Compiler and Garbage Collector]. Interpreting the bytecode, the standard impl. of JVM slows the execution of the programs. **JIT compilers interact with JVM at runtime to improve performance and compile (runtime compilation)** appropriate bytecode sequences into native machine code.

Hardware is interpreting the code instead of JVM. This leads to performance gains in the speed of execution.

JIT Compiler - The disadvantage of 'Interpreter' is that when one method is called multiple times, each time a new interpretation and a slower execution are required – this is **redundant operation**.

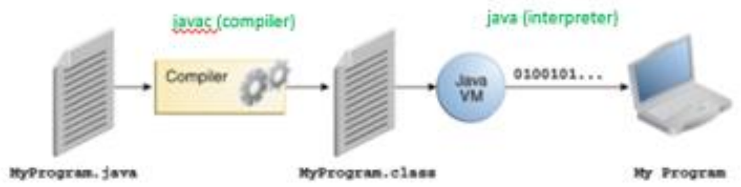

JIT Compiler
Interpreter
C1 compiler
C2 compiler
Profiler
execute the bytecode
Compile bytecode into machine code
responsible for identifying recurring code blocks
miro

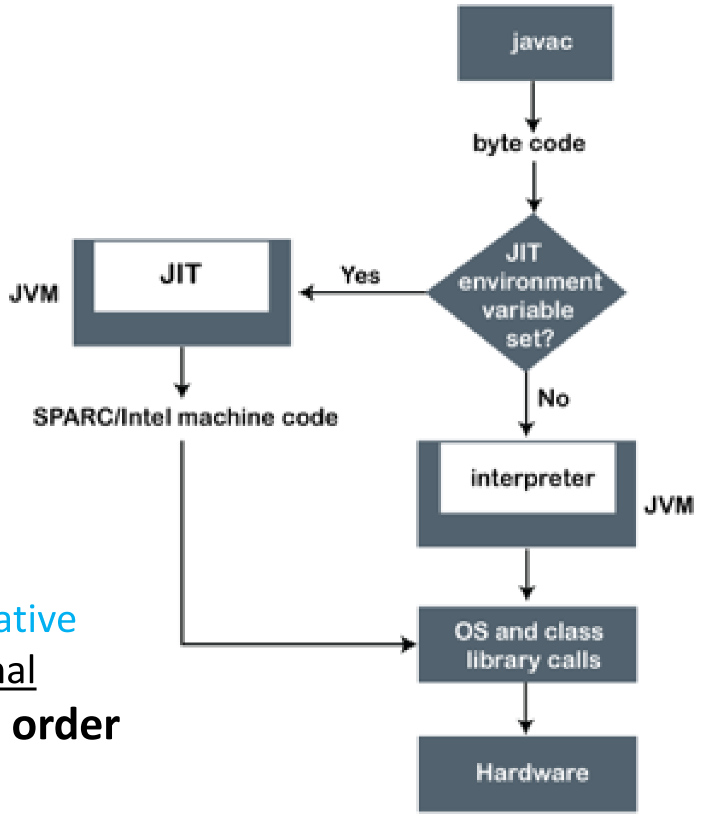
JIT Compiler **fastens it** via **compiling all bytecode (.class) into native code (machine code)** then for repeated calls it provides native code and the execution using native code is much faster than interpreting instructions one by one.
The native code is stored in the cache [**code cache** is non heap memory], thus the compiled code can be executed quicker. However, even for JIT compiler (C1, C2), it takes more time for compiling than for the interpreter to interpret. So look compiler optimizations, or GraalVM **AoT** Compiler in next slides. **Since Java 10 Graaal (alternative to C2) can** run in both mode - **JIT** and **AOT**

Java Compiler, Interpreter, and JIT

JIT Compilation Process

Overview of the Java Software development process





src-programs are compiled (**javac,** regular Java compiler) ahead of time (**AOT**) and stored as machine independent code (bytecode, *.class files, platform independent), which is then linked at run-time and executed by an interpreter. Interpreting the bytecode, the standard implementation of the JVM slows the execution of the programs.

JIT compilers (not regular compiler) interact with JVM at runtime to improve performance and compile[called runtime compilation] appropriate bytecode (JVM instruction set) sequences into native machine code that the CPU executes it directly. Default strategy used by the HotSpot during normal program execution, called **tiered compilation (C1 & C2) . It is a mix of C1 and C2 compilers in order to achieve both fast startup and good long-term performance**.

C1 (client, or **throughput** compiler), C2 (server compiler, or **optimization** compiler) JIT compilers. Not improved lately, hard-to-maintain,.. **written in C++. [GraalVM is written in Java]**

C1 handles 1,2,3 levels of compilations & optimizations, whereas C2 does Level 4.

C1 is designed to run faster and produce less optimized code, while C2, on the other hand, takes a little more time to run but produces a better-optimized code.

Since Java 10 Graal (alternative to C2) can run in both mode - **JIT** and **AOT**

Also Java 9 has AOT - https://openjdk.java.net/jeps/295

The JIT is **enabled by default**. To enable Explicitly: set JAVA_COMPILER=**jitc** or java -Djava.compiler=**jitc** <class>
To disable the JIT Set: set **JAVA_COMPILER=NONE**, Or java **-Djava.compiler=NONE** <class>

C1 and C2 Compilers

Advantages:

- JIT compilers require less memory.
- After a program has started, JIT compilers run.
- While the code is running, it is possible to optimize it.
- Any page errors can be minimized.
- On the same page, code that is used together will be localized.
- Different levels of optimization can be used.

Disadvantages:

- It can take a long time for your computer to boot up.
- **Cache (non-heap)** memory is heavily used.
- In a Java application, this increases the level of complexity.

C1 compiler handles 1,2,3 levels of compilations & optimizations, whereas C2 compiler does Level 4.

Optimization levels

The JIT compiler can compile a method at different optimization levels: **cold**, **warm**, **hot**, **very hot (with profiling)**, or **scorching**.
The hotter the optimization level, the better the expected performance, but the higher the cost in terms of CPU and memory.

- **cold** is used during startup processing for large applications where the goal is to achieve the best compiled code speed for as many methods as possible.
- **warm** is the workhorse; after start-up, most methods are compiled when they reach the invocation threshold.

The JIT compilation includes two approaches **AOT** (Ahead-of-Time compilation) and **interpretation** to translate code into machine code. AOT compiler compiles the code into a native machine language (the same as the normal compiler). It transforms the bytecode of a VM into the machine code. The following optimizations are done by the JIT compilers:

- Method In-lining
- Local Optimizations
- Control Flow Optimizations
- Constant Folding
- Dead Code Elimination
- Global Optimizations
- Heuristics for optimizing call sites

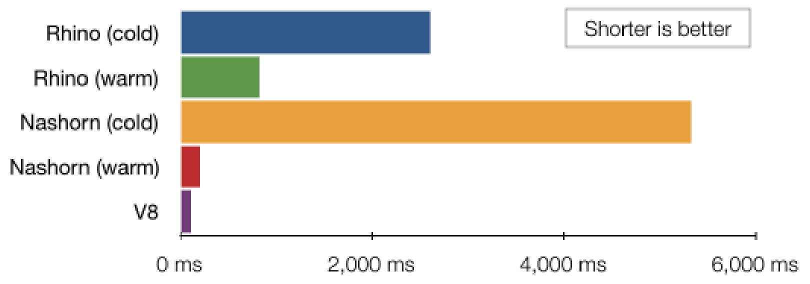
New Features in JAVA 15

Deprecations and Removals

- As a part of Java 15 removals/deprecations, Nashorn engine is removed (Nashorn supports ECMAScript 5.1 specification) …. Impacts/alternatives

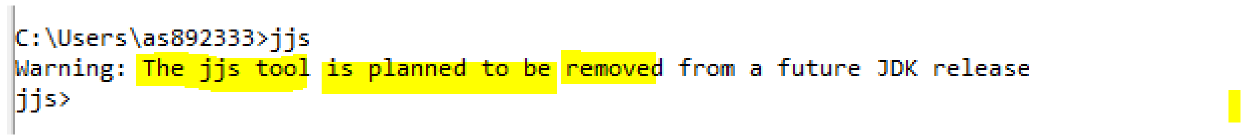
  - **Nashorn JS Engine (old one was Rhino engine)** – removed, also the tool '**jjs**' is removed. Rhino compiled all Java code into Java bytecode, produces best performance even better than C++ when running. .

This JEP also removed the below two modules:
jdk.scripting.nashorn.shell – contains the jjs tool and
jdk.scripting.nashorn – contains jdk.nashorn.api.scripting and
jdk.nashorn.api.tree packages.

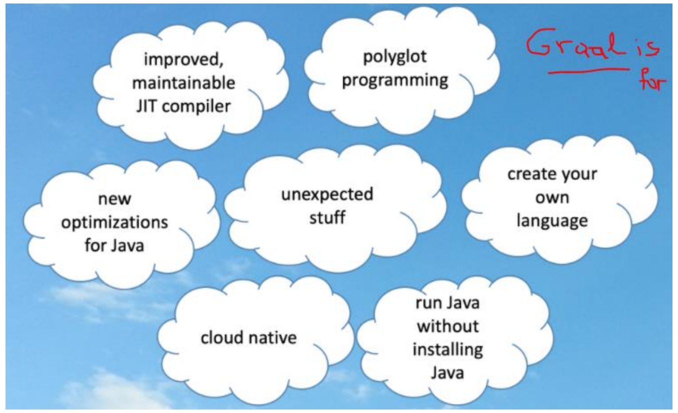


NashornInsteadOfRhino.java ? **Rhino | SpiderMonkey | V8 | Chakra | Nashorn** (rhinoceros) **| Graal (JS, NodeJS)**

JDK has (will) no JS anymore (maintenance cost, also Graal offers more, runs NodeJS, and also JS hast testing-frameworks).
See Migration Guide



```
C:\Users\as892333>jjs
Warning: The jjs tool is planned to be removed from a future JDK release
jjs>
```



improved, maintainable JIT compiler
polyglot programming
Graal is for
new optimizations for Java
unexpected stuff
create your own language
cloud native
run Java without installing Java

Alternative see: **GraalVM**, Project Detroit (V8) – bring V8 JS engine to OpenJDK

New Feature in JAVA 15

GraalVM High Performance. Cloud Native. Polyglot.

Linux AMD64 | Linux ARM64 | macOS | Windows

GraalVM is an umbrella term, consists of: **Graal** the JIT compiler, and **Truffle** (lang. runtimes), **Substrate VM** (Native Image).

Java JVM consists of: Class Loader, Runtime Memory (Data Area) and Execution Engine

GraalVM Enterprise (based on **Oracle JDK**) and **GraalVM Community** (on **OpenJDK**) editions, supports **Java 8, 11, 16. Releases:** 1-release GraalVM 19.0 is based on top of JDK version 8u212, latest-21.2.0.1

History: GraalVM has its roots in the Maxine Virtual Machine project at Sun Microsystems Laboratories (now Oracle Labs). The **goal was** removing dependency to C++ & its problems … & written in modular, maintainable and extendable fashion **in Java itself**. Moreover you can deeply understand, learn it looking the code as well.

Project goals (improve performance, reduce startup time, native images, extending JVM app. Or native app. Own lang.)

- To improve the performance of Java virtual machine-based languages to match the performance of native languages.
- To reduce the startup time of JVM-based apps by compiling them AOT with GraalVM Native Image technology.
- To enable GraalVM integration into the OracleDB, OpenJDK, Node.js, Android/iOS, and to support similar custom embeddings.
- To allow freeform mixing of code from any programming language in a single program, billed as "polyglot applications“.
- Create your own language. To include an easily extended set of "polyglot programming tools".

E.g: Twitter use Graal to run Scala app., Facebook uses it to accelerate its Spark workloads, …

GraalVM Architecture

Core Components

Runtimes: Java HotSpot VM, Javascript runtime, LLVM runtime.

Runtime Modes: JVM runtime mode, Native Image, Java on Truffle (on AST)

Libraries (JAR files):

- **Graal compiler**
- ***Polyglot API***

Utilities: JS REPL & JS interpreter, **lli** tool to run LLVM app., GraalVM Updater

Truffle Language Implementation framework and the GraalVM SDK

Additional Components

core installation can be extended with:

Tools/Utilities:

- *Native Image*
- *LLVM toolchain*
- *Java on Truffle*

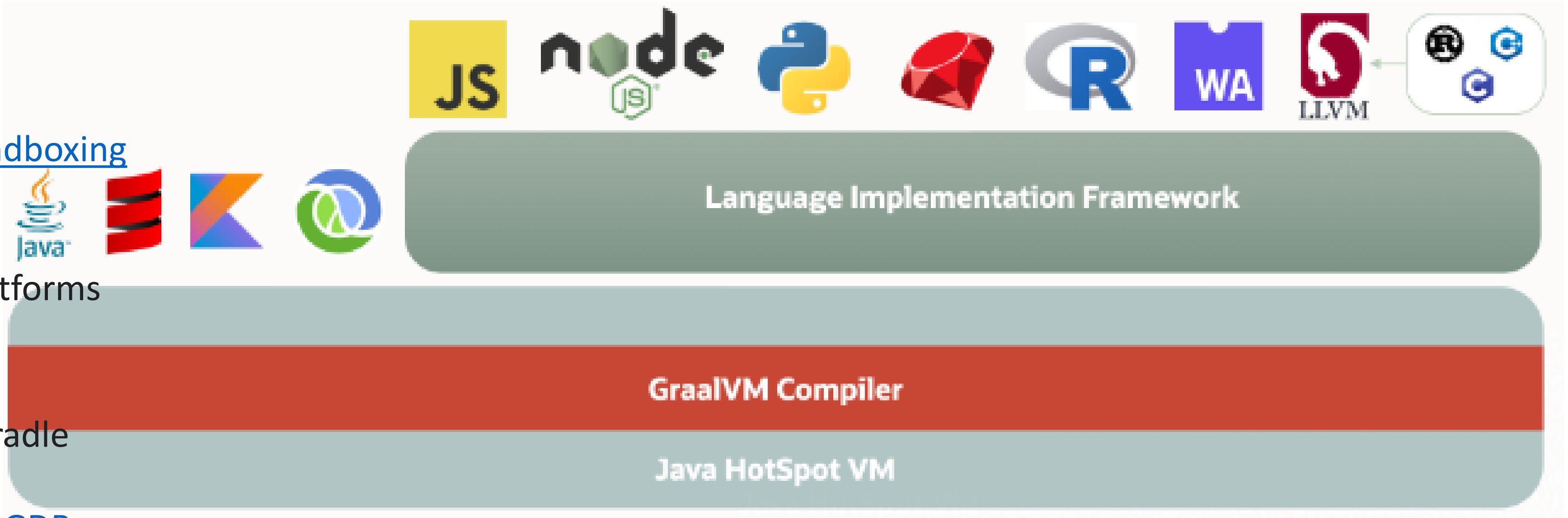
Runtimes: Node.js, Python, Ruby, R, GraalWasm(Web Assembly), Combined Languages, ..

Features

- Low Resource Usage
- Fast Startup
- Improved Security, Polyglot Sandboxing
- Compact Packaging
- Supported by Frameworks
- Supported by Leading Cloud Platforms

Using GraalVM

- For Development Maven and Gradle plugins available.
- Junit 5, Debugging via DAP and GDB
- Package&Deploy – Container Images
- Monitoring tools – VisuamVM, JFR, JMX, jvmstat, heapdumps, etc



See: Repository Structure, APIs

GraalVM Native Image is officially supported by the Fn, Gluon, Helidon, Micronaut, Picocli, Quarkus, Vert.x and Spring Boot Java frameworks

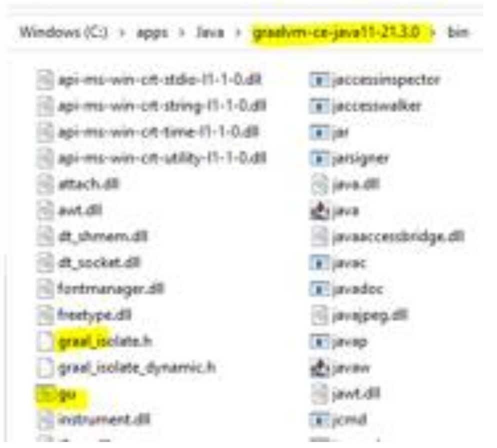
Language and Runtime Support

See: Download, APIs

GraalVM's /bin directory, as like standard JDK, with additional launchers and utilities:

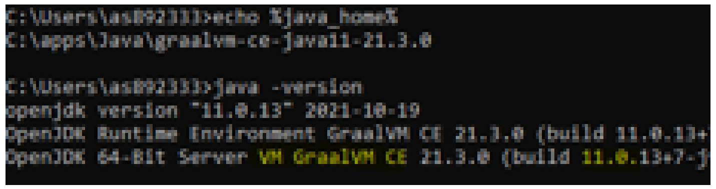
- js a JavaScript launcher
- lli a LLVM bitcode launcher
- gu the GraalVM Updater tool to install additional language runtimes and utilities

>gu list - to see installed launchers/utils



```
Windows (C:) > apps > Java > graalvm-ce-java11-21.3.0 > bin

api-ms-win-crt-stdio-l1-1-0.dll       jaccessinspector
api-ms-win-crt-string-l1-1-0.dll      jaccesswalker
api-ms-win-crt-time-l1-1-0.dll        jar
api-ms-win-crt-utility-l1-1-0.dll     jarsigner
attach.dll                            java.dll
awt.dll                               java
dt_shmem.dll                          javaaccessbridge.dll
dt_socket.dll                         javac
fontmanager.dll                       javadoc
freetype.dll                          javajpeg.dll
graal_isolate.h                       javap
graal_isolate_dynamic.h               javaw
gu                                    jawt.dll
instrument.dll                        jcmd
```



```
C:\Users\as892333>echo %java_home%
C:\apps\Java\graalvm-ce-java11-21.3.0

C:\Users\as892333>java -version
openjdk version "11.0.13" 2021-10-19
OpenJDK Runtime Environment GraalVM CE 21.3.0 (build 11.0.13+
OpenJDK 64-Bit Server VM GraalVM CE 21.3.0 (build 11.0.13+7-j
```

Besides JVM langs. Java, Scala, Kotlin, … additional languages can be supported in GraalVM based on Truffle Language Implementation framework: GraalVM JavaScript, TruffleRuby:, FastR, GraalVM Python, GraalVM LLVM Runtime , & GraalWasm

Run **Java** as it is:
>javac…. Then how to add other runtimes? Node.js, Ruby, R, WebAssembly?

Graal Compiler

Graal is a high-performance JIT compiler. See repo

- It accepts the JVM bytecode and produces the machine code.
- Uses JVMCI to communicate with the VM. **JVM Compiler Interface**: JVMCI excludes the standard tiered compilation and plug in our brand new compiler (i.e. Graal) without the need of changing anything in the JVM
- Graal compiler is alternative to C2. Unlike C2, it can run in both **JIT** and **AOT** compilation **modes** to produce native code
- It includes multiple optimization algorithms (called "Phases"), like aggressive inlining, polymorphic inlining, and others.

Advantages of writing a compiler in Java?
Safety, no crash but ex., no memory leak, tool support (debuggers, profilers). ..

To enable the use of the new JIT compiler: `-XX:+UnlockExperimentalVMOptions -XX:+EnableJVMCI -XX:+UseJVMCICompiler`

We can run a simple program in three different ways: via regular C1/C2, with the JVMCI version of Graal on Java 10, or with the GraalVM

High-performance modern Java: **E.g. Demo1, Demo2.java**
Low-footprint, fast-startup Java: Currently suffer from longer startup time and high memory usage for short-running process

Tooling support: GraalVM includes VisualVM with the standard jvisualvm command.
```
>jvisualvm &> /dev/null &
```

Extend a JVM-based application: A new **org.graalvm.polyglot** API lets you load and run code in other languages

Graal Compiler

Graph Compilation
To run guest programming languages, namely JavaScript, Ruby, R, Python, and WebAssembly in the same runtime as the host JVM-based language, the compiler should work with a language-independent intermediate representation (IR) between the source language and the machine code to be generated. A *graph* was selected for this role. The **Truffle framework code** and data (Abstract Syntax Trees) is partially evaluated to produce a **compilation graph**.

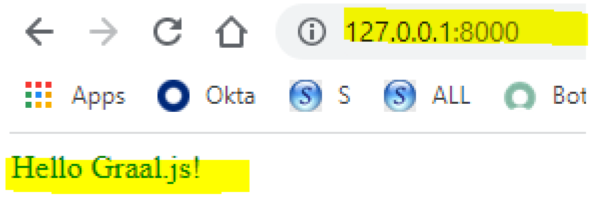
Ahead-of-time Compilation
Besides the Truffle framework, GraalVM incorporates its optimizing compiler into an advanced ahead-of-time (AOT) compilation technology – Native Image – which translates Java and JVM-based code into a native platform executable. These native executables start nearly instantaneously, are smaller, and consume less resources of the same Java application, making them ideal for cloud deployments and microservices. For more information about the AOT compilation, go to Native Image.

Compiler Operating Modes
There are two operating modes of the Graal compiler when used as the HotSpot JIT compiler: as pre-compiled machine code ("**libgraal**") [for **AOT**], or as dynamically executed Java bytecode ("jargraal"). **libgraal:** the Graal compiler is compiled ahead-of-time into a native shared library.

Language and Runtime Support

GraalVM JS Implementation provides an ECMAScript-compliant (See Kangax table) runtime to execute JS and Node.js apps. To migrate the code previously targeted to the **Nashorn** or **Rhino** engines, see migration guides.

Why GraalVM JavaScript than Nashorn? Higher performance, better tooling, and interoperability (polyglot) **It can execute Node.js apps, .. ibs..tools**


127.0.0.1:8000

Hello Graal.js!

The languages in GraalVM aim to be drop-in replacements for your existing languages. E.g. we can install a Node.js module

Run JS & Node.js. Debugging (Tooling support)

```
>C:\apps\Java\graalvm-ce-java11-21.3.0\bin>js
>1+3      or DEBUG:> js --inspect fiz.js (--inspect)
>gu install nodejs
>node -v
v14.17.6
```

100,000 libs < npm packages are compatible with GraalVM, e.g. express, react, async, mocha, etc.

```
>node validate-domain-async1.js
>npm install colors ansispan express        >npm list
>node z-node-app.js
>z-run-nodejs-serverapp →open: http://127.0.0.1:8000/
```

```
>graalpython --inspect fizzbuzz.py
>ruby --inspect fizzbuzz.rb
```

Run Python, R, LLVM Langs (C/C++, Rust, ..), ..

- **gu install R**

- gu install **python**
- graalpython

```
>1+3
> gu install llvm-toolchain    >gu install native-image
```

Language and Runtime Support

WebAssembly (Wasm) is an universal low level bytecode that runs on the web with near-native performance. It is a compilation target for languages like Rust, AssemblyScript (Typescript-like), Emscripten (C/C++), C# and much more!

Run WebAssembly

```
>gu install wasm
>emcc -o floyd.wasm floyd.c

> wasm --Builtins=wasi_snapshot_preview1 floyd.wasm
1 .
2 3 .
4 5 6 .
7 8 9 10 .
11 12 13 14 15 .
16 17 18 19 20 21 .
22 23 24 25 26 27 28 .
29 30 31 32 33 34 35 36 .
37 38 39 40 41 42 43 44 45 .
46 47 48 49 50 51 52 53 54 55 .
```

Creating HTML and JavaScript

```
//compile by Emscripten compiler, steps to perform
> emcc hello.c -s WASM=1 -o hello.html
//'emcc' is not recognized as an internal or external command
//open html in running server
```

Run WebAssembly in Java

```
> WebAssemplyByJava.java
```

Polyglot API - Combine Languages

GraalVM allows you to call one programming language into another and exchange data between them. To enable interoperability, GraalVM provides the --polyglot flag

Run Combine Languages – Polyglot (Embedding Languages). LLVM-Low Level VM

LLVM is written in C++ and is designed for compile-time, link-time, run-time, and "idle-time" optimization.
The GraalVM Polyglot API lets you embed and run code from guest languages in JVM-based host applications.
Allow interoperability with Java code the *--jvm* **flag**

JAVA[JS, Ruby, Python, LLVM…]: >Polyglot.java
JS[Java, Python, Ruby, LLVM]: >js --polyglot --jvm polyglot.js

C[…]: > gu install **llvm-toolchain**

Python[…]: > graalpython --polyglot --jvm polyglot.py

`Polyglot Options You can configure a language engine for better` **`throughput`** `or` **`startup`**

Passing options:

You can configure a language engine for better throughput or startup

- Through launchers : js, python, llvm, r, ruby
- Programmatically
- Using JVM arguments

Native Images

Native Image is a technology to compile Java code ahead-of-time to a binary – a **native executable**.

Native Image is a technology to AOT Java code to a standalone executable, called a native image. Native Image supports JVM-based languages, e.g., Java, Scala, Clojure, Kotlin. The resulting image can, optionally, execute dynamic languages like JavaScript, Ruby, R or Python . GraalVM Native Image is officially supported by the Fn, Gluon, Helidon, Micronaut, Picocli, Quarkus, Vert.x and Spring Boot Java frameworks. The **jaotc** command creates a Native Image. The experimental -XX:+EnableJVMCIProduct flag enables the use of Graal JIT

`Program Code ⇒ AST ⇒ Bytecode ⇒ Machine code (ASM)`

`Program Code → AST → Truffle → Graal → Machine code`

Native Images (read more)

```
> gu install native-image
>javac NativeImage.java
>java NativeImage
>native-image  NativeImage
```

A native executable is created by the **Native Image builder** or native-image that processes your application classes and other metadata to create a binary for a specific operating system and architecture. The native-image tool can be used to build a **native executable**, which is the default, or a **native shared library**.

> native-image [options] class [imagename] [options] //from class

- native-image [options] -jar jarfile [imagename] //from JAR file
- native-image [options] --module <module>[/<mainclass>] [options] //from module

Truffle Language Implementation Framework

In association with GraalVM, Oracle Labs developed a language abstract syntax tree interpreter called "Truffle" which would allow it to implement languages (or language-agnostic tools like debuggers, profilers, and other instrumentations) on top of the GraalVM. The Truffle framework and its dependent part, GraalVM SDK, are released under the Universal Permissive License

Truffle is a Java library that helps you to write an *abstract syntax tree* (AST) interpreter for a language. An AST interpreter is probably the simplest way to implement a language, because it works directly on the output of the parser and doesn't involve any bytecode or conventional compiler techniques, but it is often slow.

```
<dependency>
   <groupId>org.graalvm.truffle</groupId>
   <artifactId>truffle-api</artifactId>
   <version>21.3.0</version>
</dependency>
<dependency>
   <groupId>org.graalvm.truffle</groupId>
   <artifactId>truffle-dsl-processor</artifactId>
   <version>21.3.0</version>
   <scope>provided</scope>
</dependency>
```

Instrumentation-based Tool Support

The core GraalVM installation provides a language-agnostic debugger, profiler, heap viewer, and others based on instrumentation and other VM support.

Thing To Do with GraalVM

1. High-performance modern Java

GraalVM is one compiler to rule them all, meaning that it's a single implementation of a compiler written as a library which can be used for many different things. For example we use the GraalVM compiler to compile both ***ahead-of-time[AOT]*** and ***just-in-time[JIT]***, to compile multiple programming languages, and to multiple architectures. One simple way to use GraalVM is to use it as your Java JIT compiler.

GraalVM is **written in Java**, rather than C++ like most other JIT compilers for Java. GraalVM works at the level of JVM bytecode so it works for any JVM language. We think this allows us to improve it **more quickly than existing compilers**, with powerful new optimisations such as partial escape analysis that aren't available in the standard JIT compilers for HotSpot. This can make your Java programs run significantly faster.
To run without the GraalVM JIT compiler to compare, I can use the flag -XX:-UseJVMCICompiler

2. Low-footprint, fast-startup Java

The Java platform is particularly strong for long-running processes and peak performance, but short-running processes can suffer from longer startup time and relatively high memory usage. GraalVM gives us a tool that solves this problem. One of those is to compile ***ahead-of-time***, to a native executable image, instead of compiling ***just-in-time*** at runtime.

➢ native-image --no-server --no-fallback TopTen //produces executable topten (not Java launcher)

For runtime components like the garbage collector we are running our own new VM called the **SubstrateVM**, which like GraalVM is also written in Java. It has some additional advantages over basic compilation as well in that static initializers are run during compilation, so you can reduce the work done each time an application loads. It has some additional advantages over basic compilation as well in that static initializers are run during compilation, so you can reduce the work

Thing To Do with GraalVM

3. Combine JavaScript, Java, Ruby, and R

As well as Java, GraalVM includes new implementations of JavaScript, Ruby, R and Python.Written using a new language implementation framework called ***Truffle***. When you write a **language interpreter using Truffle**, Truffle will automatically use GraalVM on your behalf to give you a JIT compiler for your language. So GraalVM is not only a JIT compiler and ahead-of-time native compiler for Java, it can also be a **JIT compiler for JavaScript, Ruby, R and Python**.

The languages in GraalVM work together — there's an API which lets you run code from one language in another. This lets you write polyglot programs — programs written in more than one language.

We then need to run node with a couple of options: --polyglot to say we want access to other languages, and --jvm because the node native image by default doesn't include more than JavaScript.

- node **--polyglot --jvm** color-server.js serving at http://localhost:8080

That's the third thing we can do with GraalVM — run programs written in multiple languages and use modules from those languages together.

4. Run native languages on the JVM

Another language that GraalVM supports is C, runs C code in the same way that it runs like JavaScript and Ruby. What GraalVM actually supports is running the output of the **LLVM toolchain** — LLVM bitcode — rather than directly supporting C. This means you can use your existing tooling with C, and also other languages that can output LLVM, such as C++, Fortran, and potentially other languages in the future.

- clang -c –**emit-llvm** gzip.c
- Lli gzip.bc –d small.txt.gz //**lli – LLVM bitcode interpreter //**gu install llvm-toolchain

Thing To Do with GraalVM

5. Tools that work across all languages
If you program in Java you're probably used to very high quality tools like IDEs, debuggers and profilers. Not all languages have these kind of tools, but you get them if you use a language in GraalVM. All the GraalVM languages (except for Java at the moment) **are implemented using the common Truffle framework.** This allows us to implement functionality like debuggers once and have it available to all languages.

```
>js --inspect fizzbuzz.js    >grallvmpython --inspect fizzbuzz.py  >ruby --inspect fizzbuzz.rb   >Rscript --inspect fizzbuzz.r
Debugger listening on port 9229. To start debugging, open the following URL in Chrome:
chrome-devtools://devtools/bundled/inspector.html?ws=127.0.0.1:9229/6c478d4e-1350b196b409
```

Another tool that you may be familiar with using already from Java is VisualVM.

```
>jvisualvm &> /dev/null &
```

If we use the GraalVM version of Ruby instead, VisualVM will recognise the Ruby objects themselves. We need to use the --jvm command to use VisualVM, as it doesn't support the native version of Ruby.

```
>ruby --jvm render.rb
```

The Truffle framework is a kind of nexus for languages and tools. If you program your languages using Truffle, and you program your tools like this debugger against Truffle's tool API, then each tool works with each language, and you only have to write the tool once.

Thing To Do with GraalVM

6. Extend a JVM-based application
As well as being usable as standalone language implementations, and together in a polyglot use case, these languages and tools can also be embedded in your Java application. A new org.graalvm.polyglot API lets you load and run code in other languages and to use values from them.

7. Extend a native application
GraalVM already includes one native library built like this — it's a library that lets you run code written in any GraalVM language from native applications. **JavaScript runtimes like V8, and Python interpreters like CPython**, are often embeddable meaning that they **can be linked as a library into another application**. GraalVM lets you use any language in an embedded context, by linking to this one polyglot embedding library.

```
$ gu install --force --file native-image-installable-svm-svmee-java8-darwin-amd64-19.3.0.jar
$ gu rebuild-images libpolyglot
```

8. Java code as a native library
Java has a great ecosystem of many very high quality libraries, which often aren't available in other ecosystems, including native applications as well as other managed languages. **GraalVM lets you take Java library, either off-the-shelf or one you've written yourself, and compile it to a standalone native library** for use from other native languages. As with our native compilation before, they don't require a JVM to run.

Thing To Do with GraalVM

9. Polyglot in the database
One application of the polyglot library for embedding languages is in the Oracle Database. We've used it to create the Oracle **Database Multilingual Engine (MLE)** which includes support for using GraalVM languages and modules from SQL.

10. Create your own language
Oracle Labs and our academic collaborators have been able to make new high-performance implementations of JavaScript, R, Ruby, Python and C with a relatively small team because we have developed the Truffle framework to make this easy.

Truffle is a Java library that helps you to write an *abstract syntax tree* (AST) interpreter for a language. An AST interpreter is probably the simplest way to implement a language, because it works directly on the output of the parser and doesn't involve any bytecode or conventional compiler techniques, but it is often slow. We have therefore combined it with a technique called *partial evaluation*, which allows Truffle to use the GraalVM compiler to automatically provide a just-in-time compilation for your language, just based on your AST interpreter.

THANK YOU

References

https://github.com/oracle/grail, https://www.graalvm.org/docs/introduction/#graalvm-architecture
https://www.graalvm.org/reference-manual/ https://medium.com/graalvm/graalvm-in-2018-b5fa7ff3b917
https://www.devonblog.com/software-development/graalvm-a-polyglot-and-faster-virtual-machine-getting-started/
https://www.baeldung.com/graal-java-jit-compiler https://www.graalvm.org/reference-manual/js/FAQ/
https://www.baeldung.com/jvm-tiered-compilation https://www.beyondjava.net/category/graalvm
https://www.graalvm.org/docs/getting-started/ https://www.devonblog.com/software-development/graalvm-a-polyglot-and-faster-virtual-machine-getting-started/