

Java Runtime Images

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

[https://en.wikipedia.org/wiki/Java version history](https://en.wikipedia.org/wiki/Java_version_history)

Azat Satklichov

azats@seznam.cz,

<http://sahet.net/htm/java.html>

Compare JDK 8 and JDK 9

Before Modular JDK

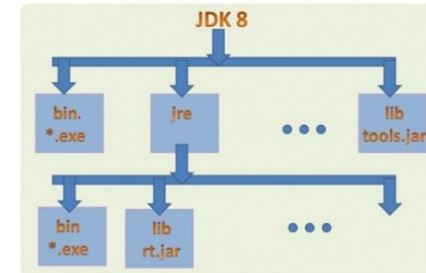
- one **rt.jar** (20 yrs. Until 2017)
- Too big, heavy (entangled classes), technical depth.
- must be backward-compatible, internal classes
- standard&ee mix)
- CLASSPATH problems (missing jars, jar hells, ...), JAR, WAR, EAR, ...
- Java 8 compact1|2|3 (tries to reduce, but still BIG). Same Docker Java Images but still runtime size is big. Also project OSGi is a way of Modularization, but it is based on current Java. So, Best option is Jigsaw JAVA.

Modular JDK

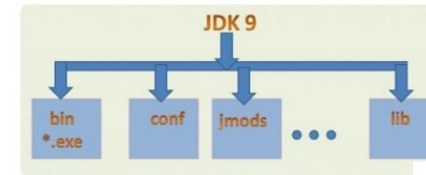
- **JDK modularized, modular apps (than monolithic)**
- Create own modules (modularize apps)
- Using module system is optional (**unnamed module**)

rt.jar

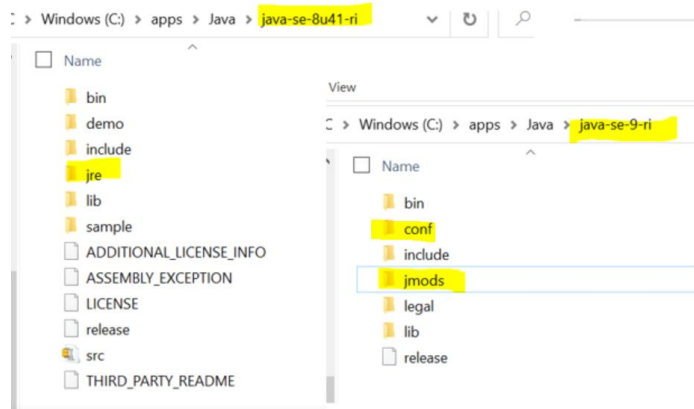
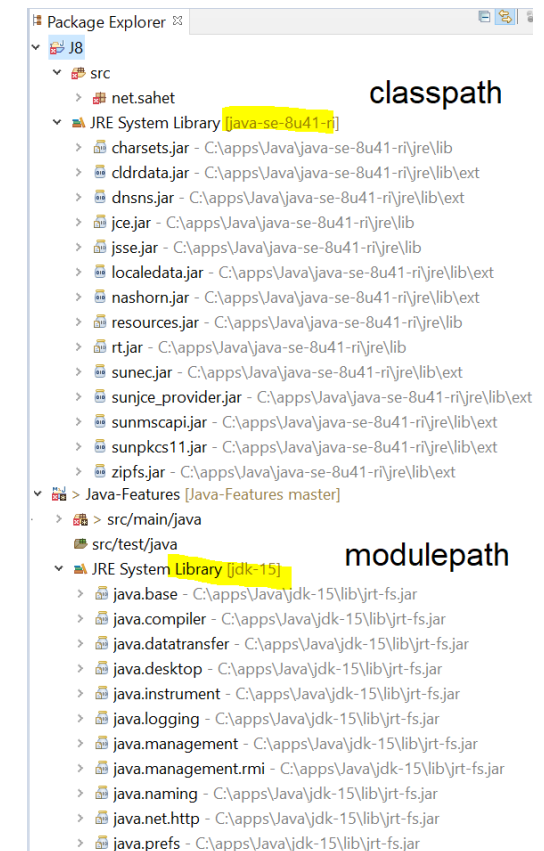
JDK 8 Folder Structure:



JDK 9 Folder Structure:



Here JDK 9 does NOT contain JRE. In JDK 9, JRE is separated into a separate distribution folder. JDK 9 software contains a new folder "jmods". It contains a set of Java 9 Modules as shown below.



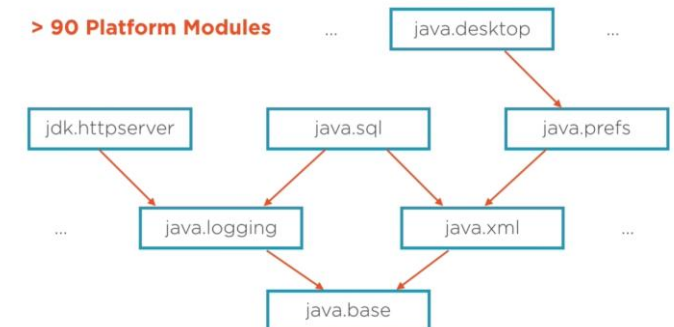
Accessibility (JDK 1 – JDK 8)

- public
- protected
- package
- private

Accessibility (JDK 9)

- public to everyone
- public but only to friend modules
- public only within a module
- protected
- package
- private

The Modular JDK: Explicit Dependencies



All modules has **implicitly** dependency to **java.base** (foundation).
E.g. Remember java Object class.

Modular JAVA (Project Jigsaw [J7])

Jigsaw project is going to introduce completely new concept - **Java Module System**.

JEPs: 200 (modular JDK), 201(modular src-code [rt.jar is too big]), 220(modular runtime images), 260(encapsulate most internal APIs), 261(module system - mod. app), 282(jlink - java linker)

[JSR 376](#) (Jigsaw - Java Platform Module System)

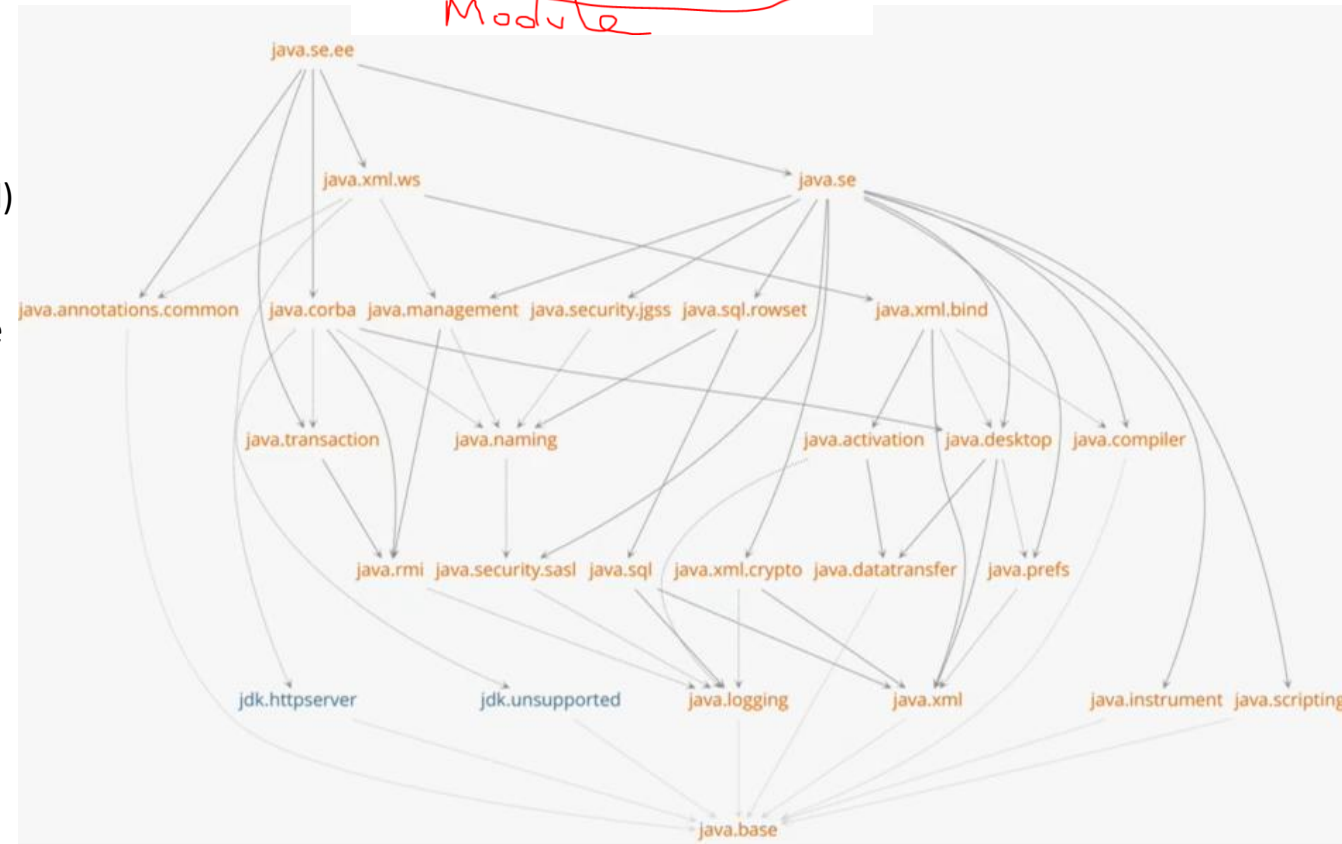
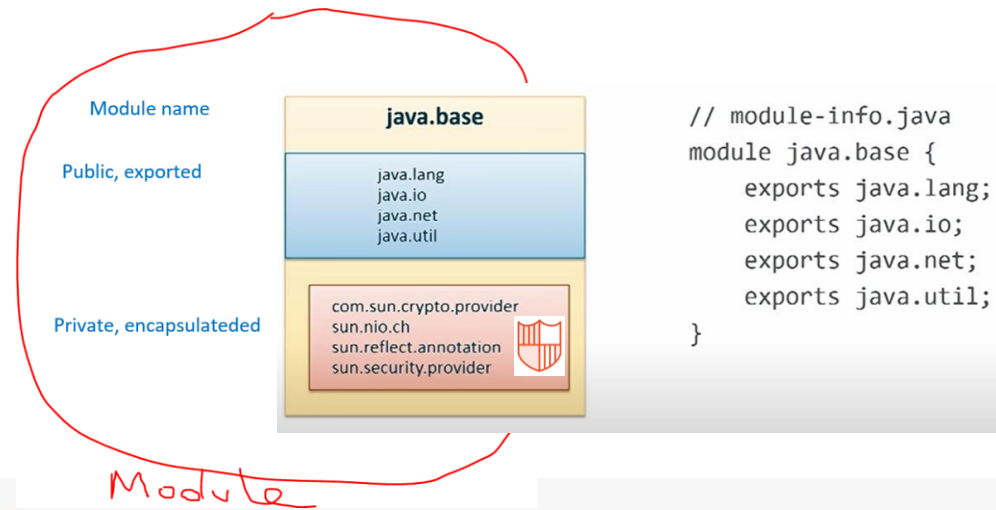
//Java 9 is modular, 11 - JEE modules removed

- **Module system** (module is named, groups related code [data+resources, mod-desc) & self sufficient [supports SRP], designed for re-use)
- **Small** and clear, compact runtime images, reliable config (**no cyclic dep.**, explicit dependency info can be used in compile and runtime)
- **Increased security (can't use internals)**, improved **performance**
- 95 < modules (jdk.*, java.* [java spec modules], java.base – default to all)
- Easy of deprecation (java.corba, ...), Future **proof** (ship, try incubators)
- **Why jar not** enough (has name but no meaning at Runtime, groups code but no encapsulation, not self contained - lack of explicit dependency in Java [yes by maven OK])
- **3 cornerstones of modularity:** Stronger encapsulation, reliable-config (clear explicit dependencies - via requires clause), well-def. interfaces)
- **Module Types:** System, Application, Automatic, Unnamed

➤ `java --list-modules` //system modules

➤ `java --describe-module java.sql`

//module-info.java (module descriptor)



Creating Modules

Module contains: one module (cannot have sub-module/s), has unique module name, packages, types, native code (if module has JMOD format), resources and one module descriptor
//module-info.java - contains module meta-data

A [module descriptor](#) describes a named module and defines methods to obtain each of its components. Modules export packages (API Interface) and require other modules (explicit dependency). Not listed means (hidden) strong encapsulation.

Modules can be distributed one of two ways: as a JAR file or as an “exploded” compiled project.
Format: jmod, jar. Jmod – contains native libs. etc... when shipping module with native code.

Creating Runtime IMAGES Packing a Java Module as a Standalone Application

Using [jlink](#) tool you can package a Java module along with only required modules (recursively) and the Java Runtime Environment into a standalone application. No Java pre-installation needed to run the application, Java is included in pack. Very small, fast and portable. Compact(custom) runtime image you can deploy to cloud..., or make [docker](#) images. (classic way: still some java image+app.)

```
> jlink --module-path "out;C:\apps\Java\jdk-17\jmods"
--add-modules test.core.modul --output out-standalone
```

Also can be optimized: `--strip-debug --compress 0|1|2`

```
> cd C:\workspace-JavaNew\J8\out-standalone
echo "Running the image"
bin\java --module test.core.modul/test.core.modul.Fibo

> bin\java --list-modules ;)
```



```
module coremodul {
    exports net.modul.core;
}

exports
opens
provides
requires
uses
```

It is common to only have one Java module per project.

```
//inspect module definitions
C:\workspace-JavaNew\Java-Features>java --describe-module java.sql
java.sql@17-ea
exports java.sql //package name
exports javax.sql
requires java.transaction.xa transitive
requires java.base mandated
requires java.logging transitive
requires java.xml transitive
uses java.sql.Driver
```

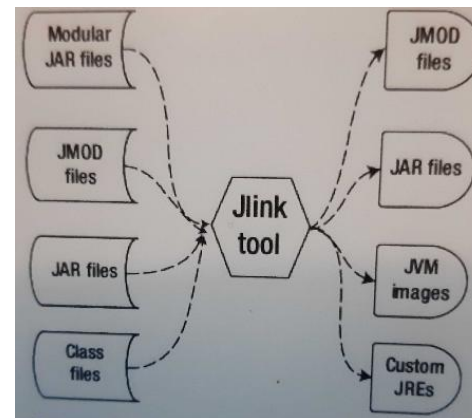
//module name



```
C:\workspace-JavaNew\J8\out-standalone>bin\java --list-modules
java.base@17
test.core.modul

C:\workspace-JavaNew\J8\out-standalone>
```

JDK17 (without app): 293.8 MB
Runtime Image (app+runtime): 40.3 MB
Runtime Image (app+runtime) optimized: 25 MB



Migrating Classpath based apps to Module based (modulepath) - Concerns

JAVA 8 → 9 (that easy?) > javac -cp \$CLASSPATH, java -cp \$CLASSPATH

using module system is optional (**unnamed module, automatic m.**), can keep using classpath.

The **unnamed module** can read all named modules, **automatic** or non-automatic.

//unless use JDK types 1. encapsulated, 2. non-default java modules. 3. Cyclic, etc.

Automatic module - add any library's JAR file to an app's module path, it becomes automatic

1. Using encapsulated types

```
import sun.security.x509.X500Name;
```

```
public class MigrationExample { public static void main(String[] args) { X500Name nem = new X500Name("CN=user"); }}
```

//apps still allowed to use encapsulated types in JDK for backward comp.

//Runs on Java 8, but strong encapsulation make it fail in Java 9

If you want to focus on future create modular applications

- Code must use public APIs instead of encapsulated internal APIs, use tool **jdeps**
- Run app with:> **-illegal-access=deny** [**permit** /**default**] so this is future-proof [**not for future, has strong encapsulation, note that =permit may be removed in future**]

If you don't want to change the code

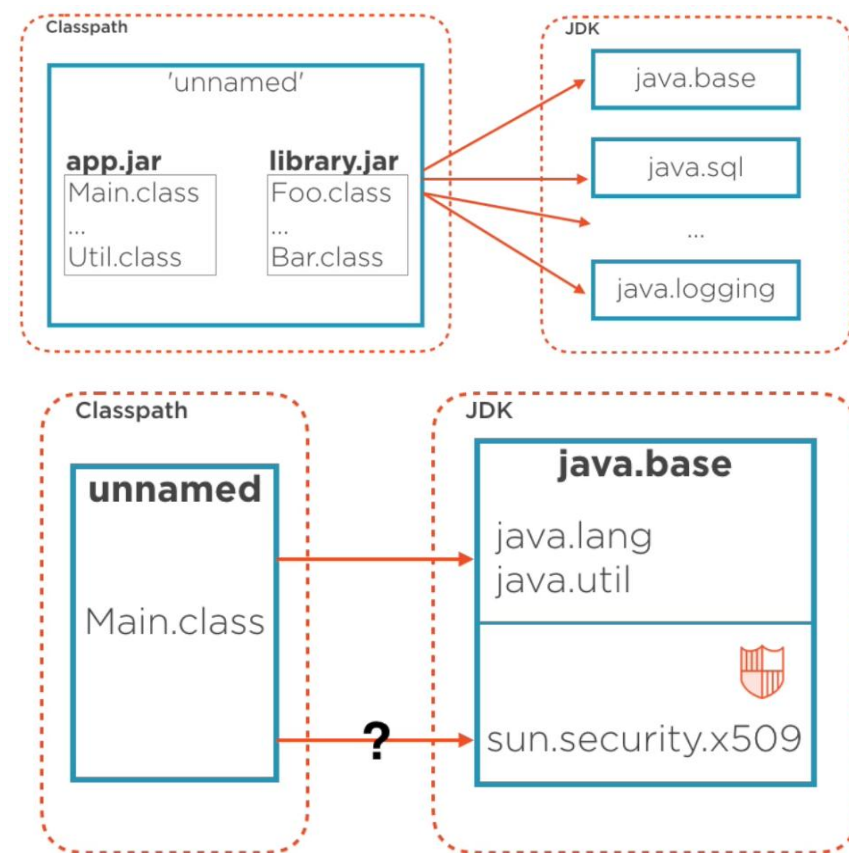
> javac --add-exports java.base/sun.security.x509=ALL-UNNAMED MigrationExample.java

MigrationExample.java:3: warning: X500Name is internal proprietary API and may be removed in a future release

```
import sun.security.x509.X500Name;
```

> java --add-exports java.base/sun.security.x509=ALL-UNNAMED MigrationExample

➤ jdeps -jdkinternals MigrationExample.class



```
C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>java -version
openjdk version "1.8.0_41"
OpenJDK Runtime Environment (build 1.8.0_41-b04)
OpenJDK Client VM (build 25.40-b25, mixed mode)

C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>javac MigrationExample.java
MigrationExample.java:3: warning: X500Name is internal proprietary API and may be removed in a future release
import sun.security.x509.X500Name;
```

```
C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>java -version
openjdk version "11" 2018-09-25
OpenJDK Runtime Environment 18.9 (build 11+28)
OpenJDK 64-Bit Server VM 18.9 (build 11+28, mixed mode)

C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>javac MigrationExample.java
MigrationExample.java:3: error: package sun.security.x509 is not visible
import sun.security.x509.X500Name;
^
(package sun.security.x509 is declared in module java.base, which does not export it to the unnamed module)
1 error
```


Migrating Classpath based apps to Module based (modulepath) - Concerns

2. Using non-default Java SE modules

```
import javax.xml.bind.DatatypeConverter;

public class MigrationExample2 {

public static void main(String[] args) throws IOException {
    DatatypeConverter.parseBase64Binary("EWsaRS43dshfJUir"); }}
```

//not reachable, because those libs are app-server specific

implementations, and even removed now, solution:

- javac --add-modules javax.xml.bind MigrationExample2.java
- java --add-modules javax.xml.bind MigrationExample2
- java --add-modules java.se.ee MigrationExample2
- jdeps MigrationExample2.class

```
<plugin>
<groupId>org.apache.maven.plugins</groupId>
<artifactId>maven-compiler-plugin</artifactId>
<version>3.8.0</version>
<configuration>
<source>9</source>
<target>9</target>
<compilerArgs>
<arg>--add-modules</arg>
<arg>javax.xml.bind</arg>
</compilerArgs>
</configuration>
</plugin>
```

```
C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>java -version
openjdk version "1.8.0_41"
OpenJDK Runtime Environment (build 1.8.0_41-b04)
OpenJDK Client VM (build 25.40-b25, mixed mode)

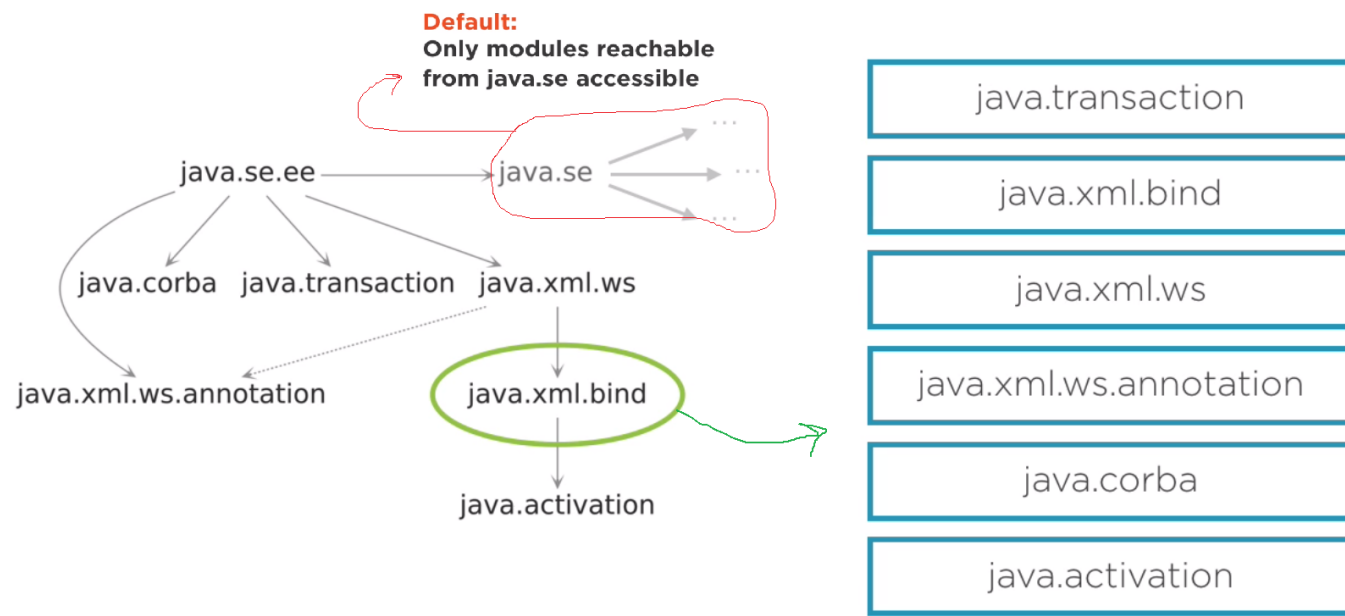
C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>javac MigrationExample2.java

C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>
```

```
C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>java -version
openjdk version "11" 2018-09-25
OpenJDK Runtime Environment 18.9 (build 11+28)
OpenJDK 64-Bit Server VM 18.9 (build 11+28, mixed mode)

C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>javac MigrationExample2.java
MigrationExample2.java:5: error: package javax.xml.bind does not exist
import javax.xml.bind.DatatypeConverter;
^
```

Using Non-default Modules



C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>jdeps MigrationExample2.class

Using New Tools in JDK 9

<https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html>

- **jdeprscan** - used to scan usage of deprecated APIs in jar file, classpath, or src-dir.

```
>jdeprscan.exe --class-path . ProcessApilImprovements_JEP102
```

```
class features/in/java9/ProcessApilImprovements_JEP102 uses deprecated method java/awt/List::addItem(Ljava/lang/String;)V
```

- **jdeps** - analysis your code base (by path to the .class file, dir, jar) list package-wise dependencies and modules. Helpful to migrate to modular-app.

```
C:\workspace-JavaNew\Java-Features\src\main\java\features\in\java9>jdeps .
```

- **jlink** - used to select modules and create a smaller runtime image with the selected modules. [jlink - Java Linker](#)

```
>jlink --module-path mods/:%JAVA_HOME/jmods --add-modules comp-packt --output img
```

- **jmod** - is a new format for packaging your modules. This format allows including native-code, config-files, and other data that do not fit into jar-file.

The Java modules are packaged as JMOD files. **jmod** command-line allows create/list/describe/hash JMOD files.

```
>jmod ..
```

jjs - -- removed in Java 15 (....)

- **Diagnostic tools:** Jcmd, jinfo, jps, jmap, jstack, jstat, <https://visualvm.github.io>
- **jshell** - JEP 222: Jshell (REPL – Read, Eval, Print, Loop) similar to NodeJS, List, Groovy, Scala has. Easily, quickly test new ideas, new features, review API, small POC. Just on CMD. No need to write java class, compile, run. Support code completion (TAB-TAB), analyze, build-in doc. Auto exception-handling on checked ones, but you can do it explicitly.

```
jshell> "aaaz c".matches("[a]+z\\sc")
```

```
/help /vars /imports /methods /types /save myCode.jsh, /open myCode.jsh , /open MyDemo.java >jshell --class-path ...
```

```
/set feedback verbose
```

```
>jshell --class-path commons-lang3-3.12.0.jar jshell> import org.apache.commons.lang3.StringUtils
```

```
jshell> StringUtils.leftPad("Broadcom", 33)
```

Jshell can be used as an API for (code completion, source code analysis, ..) IDEs can use it,. Or Jshell js = Jshell.create(); js.eval..

GraalVM



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

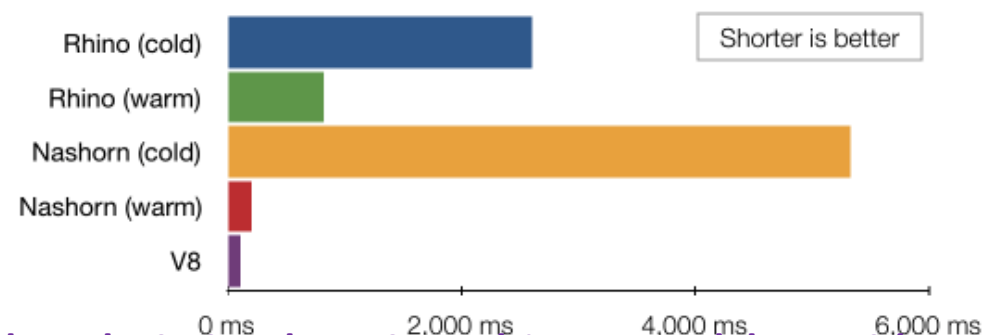
'jjs' is removed.

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 ?](#)

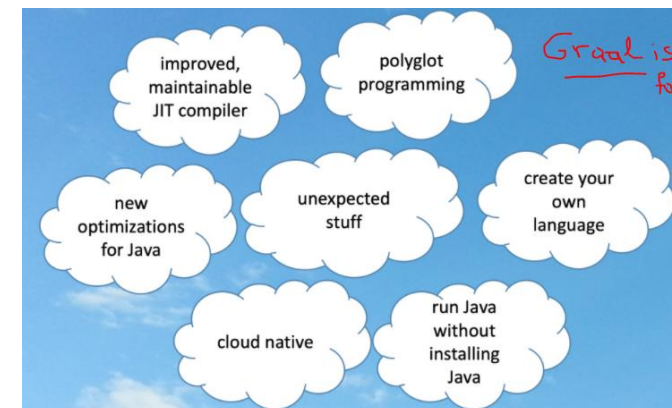
[Rhino](#) | [SpiderMonkey](#) | [V8](#) | [Chakra](#) | [Nashorn \(rhinoceros\)](#) | [Graal \(JS, NodeJS\)](#)

JDK has (will) no JS anymore (maintenance cost, also Graal offers more, and also JS fast testing-frameworks).

See [Migration Guide](#)

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

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



Java Compiler, Interpreter, and JIT

Overview of the Java Software development process



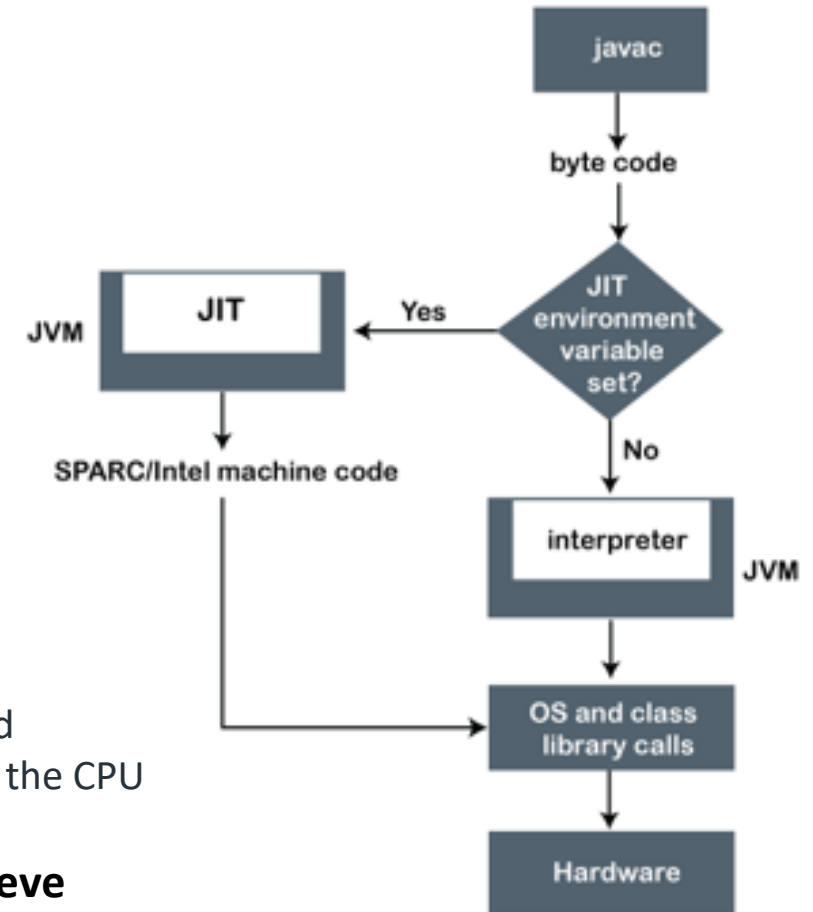
src-programs are compiled (**javac**, regular Java compiler) ahead of time (**AOT**) and stored as machine independent code (**bytecode**, *.class files, platform independent), which is then linked at run-time and executed by an **interpreter**. Interpreting the bytecode, the standard implementation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

JIT compilers (not regular compiler) interact with JVM at runtime to improve performance and compile 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), C2 (server compiler) JIT compilers. **Not improved lately, hard-to-maintain,.. written in C++**.

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

JIT Compilation Process



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

New Feature in JAVA 15



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

Linux AMD64	Linux ARM64	macOS	Windows
-------------	-------------	-------	---------

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):

- [Gaal compiler](#)
- [Polyglot API](#)

Utilities: JS REPL & JS interpreter, Ili tool to run LLVM app., [GaalVM Updater](#)

Truffle Language Impl. framework and the GraalVM SDK

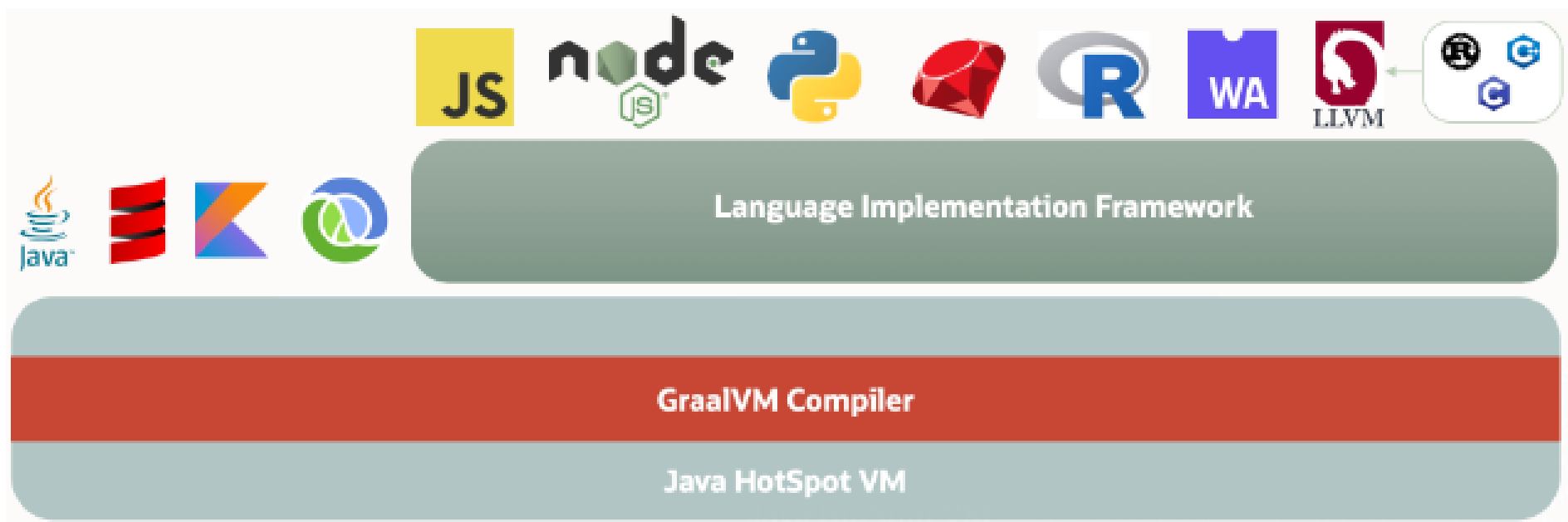
Additional Components

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, ..



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

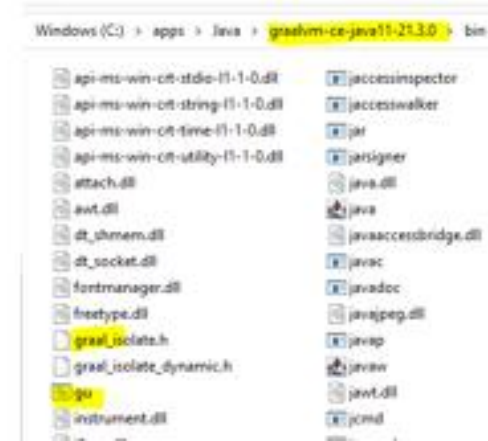
Language and Runtime Support

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/utls



```
C:\Users\asB923111>echo %java_home%
C:\apps\Java\graalvm-ce-java11-21.3.0

C:\Users\asB923111>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

Advantages of writing a compiler in Java?

Safety, no crash but ex., no mem.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

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.

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.

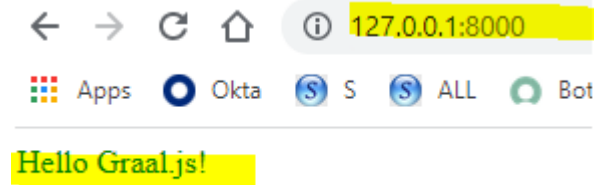
```
//run Node JS app with domains_list.txt
```

```
>node validate-domain-async1.js
```

```
>npm install colors anispan express >npm list
```

```
>node z-node-app.js →open: http://127.0.0.1:8000/
```

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



```
>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

> WebAssemblyByJava.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)

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

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 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

Native Images ([read more](#))

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

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

Program Code → AST → Truffle → Graal → Machine code

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.

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.

```
<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>
```

Spring Native

Project goals

- Those [native Spring](#) applications can be deployed as a standalone executable (no JVM installation required)

Key advantages

- Almost instant startup (typically < 100ms)
- Instant peak performance
- Reduced memory consumption
- No longer costly build times and fewer runtime optimizations than the JVM.

Scenarios where native could fit for your Spring application

- Serverless with Spring Cloud Function
- Cheaper and more sustainable hosting of your Spring microservices
- Good fit with Kubernetes platforms like [VMware Tanzu](#)
- Want to create optimal container images packaging your Spring applications and services

The goal is to incubate the support for Spring Native, an alternative to Spring JVM, and provide a native deployment option designed to be packaged in lightweight containers.

There are also some drawbacks and trade-offs that the GraalVM native project expect to improve on over time.

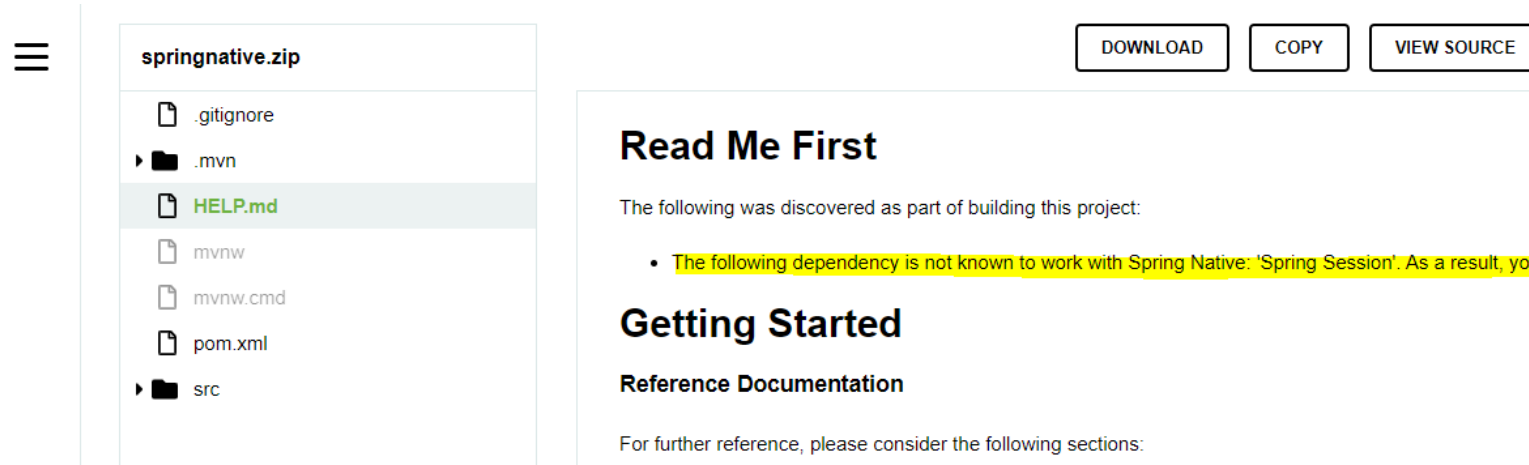
- Building a native image is a heavy process that is slower than a regular application
- A native image has fewer runtime optimizations after warmup.
- Finally, it is less mature than the JVM with some different behaviors.

The key differences between a regular JVM and this native image platform are:

- A static analysis of your application from the main entry point is performed at build time.
- The unused parts are removed at build time.
- Configuration is required for reflection, resources, and dynamic proxies.
- Classpath is fixed at build time.
- No class lazy loading: everything shipped in the executables will be loaded in memory on startup.
- Some code will run at build time.
- There are some [limitations](#) around some aspects of Java applications that are not fully supported.

Example project

Go to spring initializr: <https://start.spring.io/>



The screenshot shows the Spring Native project page. On the left, a file explorer displays the contents of 'springnative.zip', including '.gitignore', '.mvn', 'HELP.md' (highlighted), 'mvnw', 'mvnw.cmd', 'pom.xml', and 'src'. On the right, there are buttons for 'DOWNLOAD', 'COPY', and 'VIEW SOURCE'. Below these, the 'Read Me First' section states: 'The following was discovered as part of building this project:' followed by a bullet point: 'The following dependency is not known to work with Spring Native: 'Spring Session'. As a result, your application may not work as expected.'

Spring Native [Experimental]

DEVELOPER TOOLS

Incubating support for compiling Spring 2.x applications to native executables using the GraalVM native-image compiler. **Not needed with Spring Boot 3.**

each Spring Native version only supports a specific Spring Boot version – for example,

- Spring Native 0.10.0 supports Spring Boot 2.5.1.
- Spring Native 0.12.1 supports Spring Boot 2.7.4.

Make sure to check the generated `HELP.md` file

Using `mvn spring-boot:build-image` or `gradle bootBuildImage`, you can generate an optimized container image (minimal OS layer, a small native executable - required JDK part, Spring, and other dependencies for app.)

e.g. Minimal container image with a 65MB [executable containing Spring Boot, Spring MVC, Jackson, Tomcat, the JDK and the app.]

>`mvn spring-boot:build-image`

```
2022-10-20 18:24:04.374 INFO 15640 --- [main] o.s.a.build.ContextBootstrapContributor : Processed 55 bean definitions in 4197ms
2022-10-20 18:24:05.014 INFO 15640 --- [main] o.s.nativex.support.SpringAnalyzer : Spring Native operating mode: native
```

Modules

Spring Native is composed of the following modules:

- **spring-native**: for running Spring Native, provides also [Native hints](#) API.
- **spring-native-configuration**: configuration hints for Spring classes used by Spring AOT plugins
- **spring-native-docs**: reference guide, in asciidoc format.
- **spring-native-tools**: tools used for reviewing image building configuration and output.
- **spring-aot**: AOT generation infrastructure common to Maven and Gradle plugins.
- **spring-aot-test**: Test-specific AOT generation infrastructure.
- **spring-aot-gradle-plugin**: Gradle plugin that invokes AOT generation.
- **spring-aot-maven-plugin**: Maven plugin that invokes AOT generation.
- **samples**: contains various samples that demonstrate features usage and are used as integration tests.

How to Build Images

There are two main ways to build a Spring Boot native application:

- Using [Spring Boot Buildpacks support](#) to generate a lightweight container containing a native executable.
- Using [the Native Build Tools](#) to generate a native executable

Buildpacks

spring-native provides native configuration APIs like `@NativeHint` as well as other mandatory classes required to run a Spring application as a native image.

```
<dependency>
<groupId>org.springframework.experimental</groupId>
<artifactId>spring-native</artifactId>
<version>${spring-native.version}</version>
</dependency>
```

...

```
<image>
<builder>paketobuildpacks/builder:tiny</builder>
<env>
<BP_NATIVE_IMAGE>true</BP_NATIVE_IMAGE>
</env>
</image>
```

... Spring ahead-of-time (AOT) [Maven plugin perform AOT](#) transformations on app. required to improve native image compatibility and footprint.

```
<groupId>org.springframework.experimental</groupId>
<artifactId>spring-aot-maven-plugin</artifactId>
<version>${spring-native.version}</version>
```

If you want to change the default native image toolkit used by Buildpack (Liberica NIK) to an alternative one

```
<buildpack>gcr.io/paketo-buildpacks/graalvm</buildpack>
```

```
<buildpack>gcr.io/paketo-buildpacks/java-native-image</buildpack>
```

tiny builder allows small footprint and reduced surface attack, you can also use base (the default) or full builders to have more tools available in the image for an improved developer experience.

GraalVM Native Build Tools

As an alternative to the Paketo buildpacks, we can use [GraalVM](#)'s [native build tools](#) to compile and build native images using GraalVM's *native-image* compiler

```
> sdk install java 21.0.0.2.r8  
> gu install native-image
```

... a profile named *native* with build support of a few plugins like [native-maven-plugin](#) and *spring-boot-maven-plugin*:

```
<id>native</id> <build> <plugins> <plugin>  
<groupId>org.graalvm.buildtools</groupId>  
..
```


Ahead-of-time transformations

[NativeConfiguration](#) and others [dynamic configuration mechanisms](#) allow more powerful and dynamic configuration generation, but beware their APIs will evolve a lot in upcoming versions

The last and probably most powerful mechanism available when working with the ahead-of-time transformation system is the capability to automatically generate native-optimized code (source and bytecode) using the closed-world assumptions introduced by Spring Boot deployment model combined with GraalVM native image characteristics.

For example, for each class annotated by [@Controller](#), an entry will be added to a generated [reflect-config.json](#) file.

Some native configuration can not be inferred, for those cases we are introducing [native hint annotations](#) (see [Javadoc](#))

E.g. PostgreSQL driver support with Spring Native provides hints that will allow generation of the right entries in native image [reflect-config.json](#), [resource-config.json](#) and [native-image.properties](#)

Creating Docker Images with Spring Boot

Creating Docker Images with Spring Boot



THANK YOU

References

<https://www.baeldung.com/oracle-jdk-vs-openjdk>

<https://medium.com/@javachampions/java-is-still-free-c02aef8c9e04>

<https://www.journaldev.com/13106/java-9-modules>

<https://mkyong.com/java/what-is-new-in-java-15/>

<https://blogs.oracle.com/javamagazine/post/java-project-amber-lambda-loom-panama-valhalla>

<https://docs.oracle.com/en/java/javase/17/language/java-language-changes.html#GUID-67ED83E7-D79F-4F46-AA33-41031E5CD094>