Learn GraalVM™

GraalVM™

eazy bytes

GraalVM is a high performance JDK that speeds up the performance of Java and JVM-based applications and simplifies the building and running of Java cloud native services. The optimized compiler generates faster code and uses fewer compute resources, enabling applications to start instantly.

Why we need to use GraalVM, when I am good with traditional JDKs ?

- Businesses want to deliver innovative software that scales and automates processes, while also running faster and consuming fewer resources.

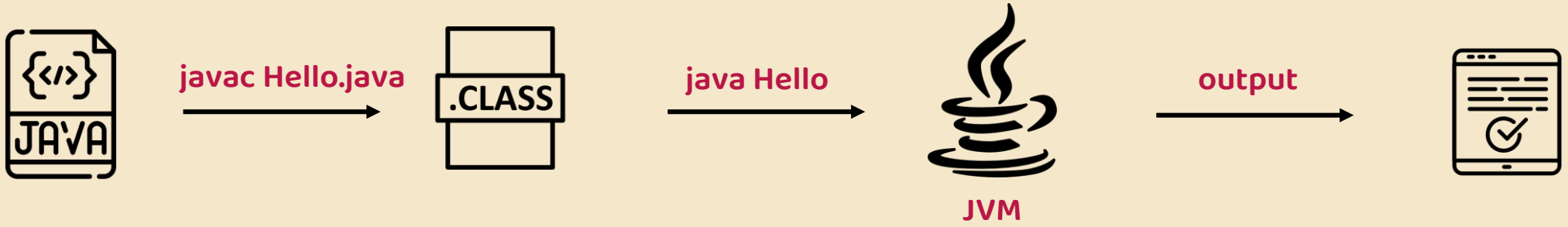
- Organizations are moving applications to the cloud and adopting cloud-native architectures like microservices, serverless to reduce costs and improve efficiency.

GraalVM, built on trusted Oracle Java SE, solves the challenges of cloud-native application development by providing a state-of-the-art JIT compiler that accelerates application performance, lowers required memory per operation, and reduces operating costs.

In simple words, GraalVM enhances the Java ecosystem by offering a compatible and better-performing Java Development Kit (JDK).

What happens inside normal JDKs ?

eazy bytes



What happens inside JVM ?

- **Class Loading:** The JVM starts by loading the necessary classes and resources required by the Java program. It searches for bytecode files (.class) that correspond to the classes used in the program and loads them into memory.

- **Bytecode Verification:** Once the classes are loaded, the JVM performs bytecode verification. It checks the bytecode for any violations of the Java language specification, such as illegal type casts or access to private fields. This step ensures the safety and security of the JVM by preventing the execution of malicious or malformed bytecode.

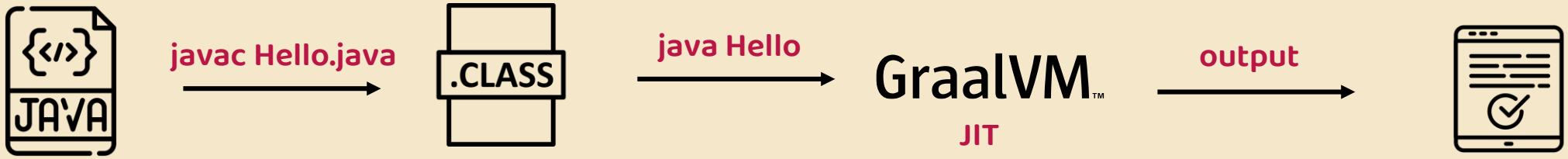
- **Just-In-Time (JIT) Compilation :** After the bytecode is verified, the JVM employs a Just-In-Time (JIT) compiler to convert the bytecode into machine code. The JIT compiler analyzes the bytecode and identifies portions of the code that are frequently executed, known as hot spots. It optimizes these hot spots by translating them into highly optimized machine code, specific to the underlying hardware and operating system. This compilation process is known as dynamic or on-the-fly compilation because it happens during runtime.

- **Execution:** Once the JIT compilation is complete, the JVM starts executing the optimized machine code. It interprets the bytecode and executes the corresponding native instructions. At this stage, the Java program is running natively on the underlying hardware, similar to programs written in other programming languages.

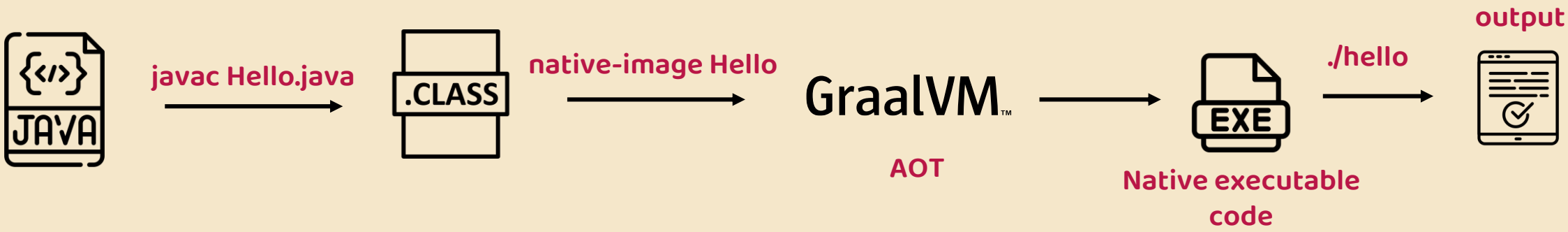
What happens with GraalVM

eazy bytes

JIT mode



AOT mode



GraalVM includes a Native Image utility to compile applications ahead-of-time (AOT) into native executables. The executables are smaller, start nearly instantaneously, and consume a fraction of the compute resources needed to run the same Java application on a Java Virtual Machine. It makes GraalVM ideal for cloud deployments and microservices. The native executables are comparable to native C/C++ programs in startup, memory usage, and performance, without leaving the powerful ecosystem of Java.

What happens in GraalVM AOT mode ?

eazy bytes

Application classes

Libraries

JDK

Substrate VM

native-image

Iterative static analysis of the code will be done by GraalVM until a fixed point reached

GraalVM scans all the reachable classes, methods etc. Any unused code will be ignored

Any dynamic code invoked with Java reflections, JNI, Dynamic proxy etc. can't be analysed by GraalVM since it performs static analysis. To over come this, we need to provide Reachability Metadata to GraalVM

Ahead-of-Time Compilation

Native executable code

Since lot of iterative analysis is going to happen, the native image generation process require more time and computational effort compared to traditional compilation/build process.

Advantages of GraalVM

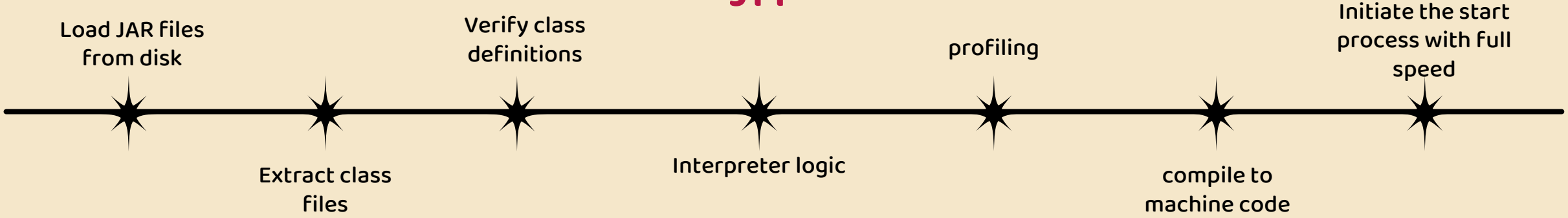
eazy bytes

Apps Start instantly

AOT

Since the number of steps with GraalVM is less and code is ready for execution in the native format, the start time will be less compared to traditional JDKs.

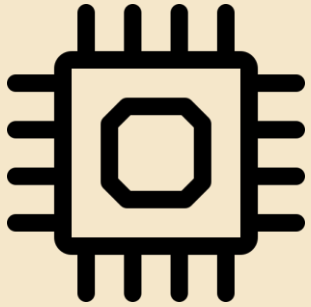
JIT



Advantages of GraalVM

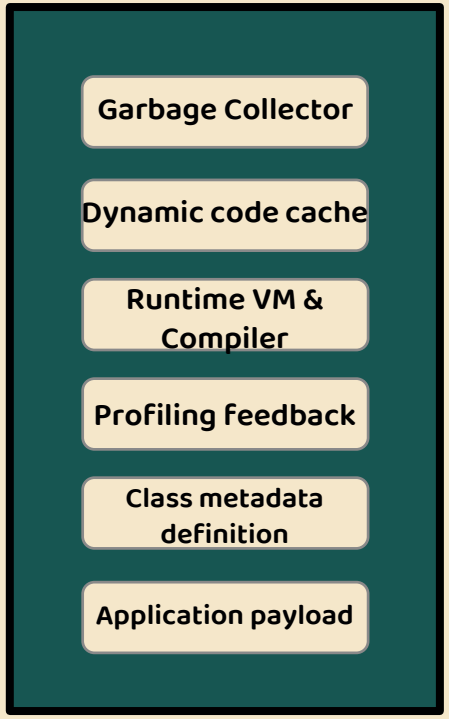
eazy bytes

Low memory consumption

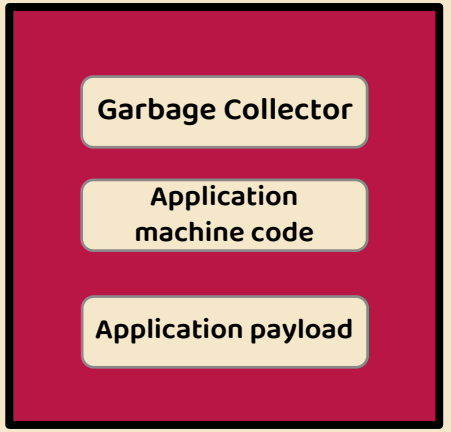


Compact packaging

JIT



AOT



Since the number of activities at run time by AOT are less, it will consume less memory compared to JIT

Since AOT doesn't require traditional JVM component and ignores unreachable code (dead code) the size of the AOT generated executable code will be less

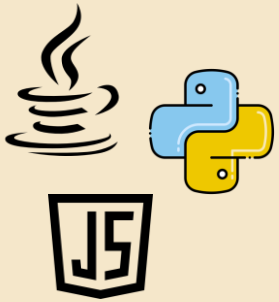
Advantages of GraalVM

eazy bytes

Reduced Attack Surface

- Only code reachable by the application are included in the image and no new code can be loaded at run time.
- Reflection, Deserialization etc. are disabled and require explicit Reachability metadata configuration

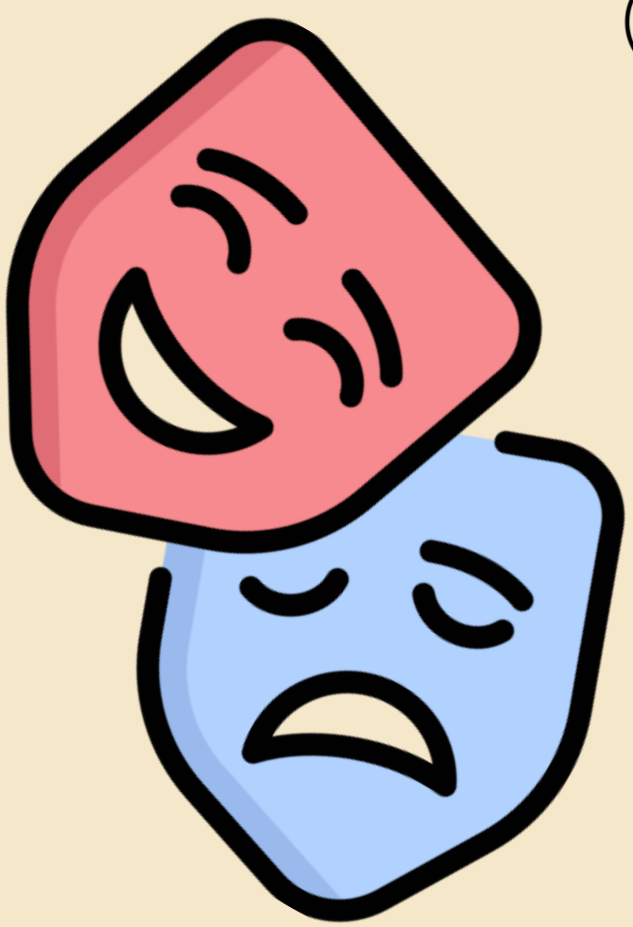
Polyglot support



- GraalVM provides support all JVM languages like Java Kotlin, Scala, Groovy, non JVM languages like Python, R, Ruby, JavaScript and LLVM languages like C, C++.
- GraalVM adds ability to efficiently mix multiple languages in a single polyglot application, and execution speeds that typically exceed those of the original language runtimes.

"Is GraalVM too good to be true?"

eazy bytes



GraalVM static analysis may not automatically detect all your reflection code, JNI, Dynamic proxy code. Workaround is to provide this information manually to the GraalVM using Reachability metadata

Maintining Reachability metadata for our own application code is fine, but how about the dynamic code used in libraries, frameworks ?

The solution is https://github.com/oracle/graalvm-reachability-metadata , to share and reuse metadata for libraries and frameworks in the Java ecosystem. Apart from this, major frameworks like Spring Boot, Quarkus, Micronaut, Helidon are already updated their code to provide this Reachability metadata.

Need lot of computational efforts at build time. Make sure to use a powerful machine with the same platform target and OS. You can't build on windows and deploy the executable code onto Linux server

Duing local development and unit testing, it will be annoying to use AOT because it takes lot of time to generate native executable code. Instead use JIT and go for AOT only during the final build process