Ausgabe 02/2020

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

D: 4,90 EUR | CH: 9,80 CHF | A: 5,60 EUR | Benelux: 5,80 EUR

4 191978 304903 02

# Java aktuell

iJUG Verbund
www.ijug.eu

**High Performance**
Ein Blick hinter die Kulissen von Java

**Immutability**
Vorteile von unveränderlichen Datenstrukturen

**Zeitreise**
Die Highlights der Java-Versionen 8 bis 13 im Überblick

# Das Herz des Java-Universums

DW *Made for minds.*

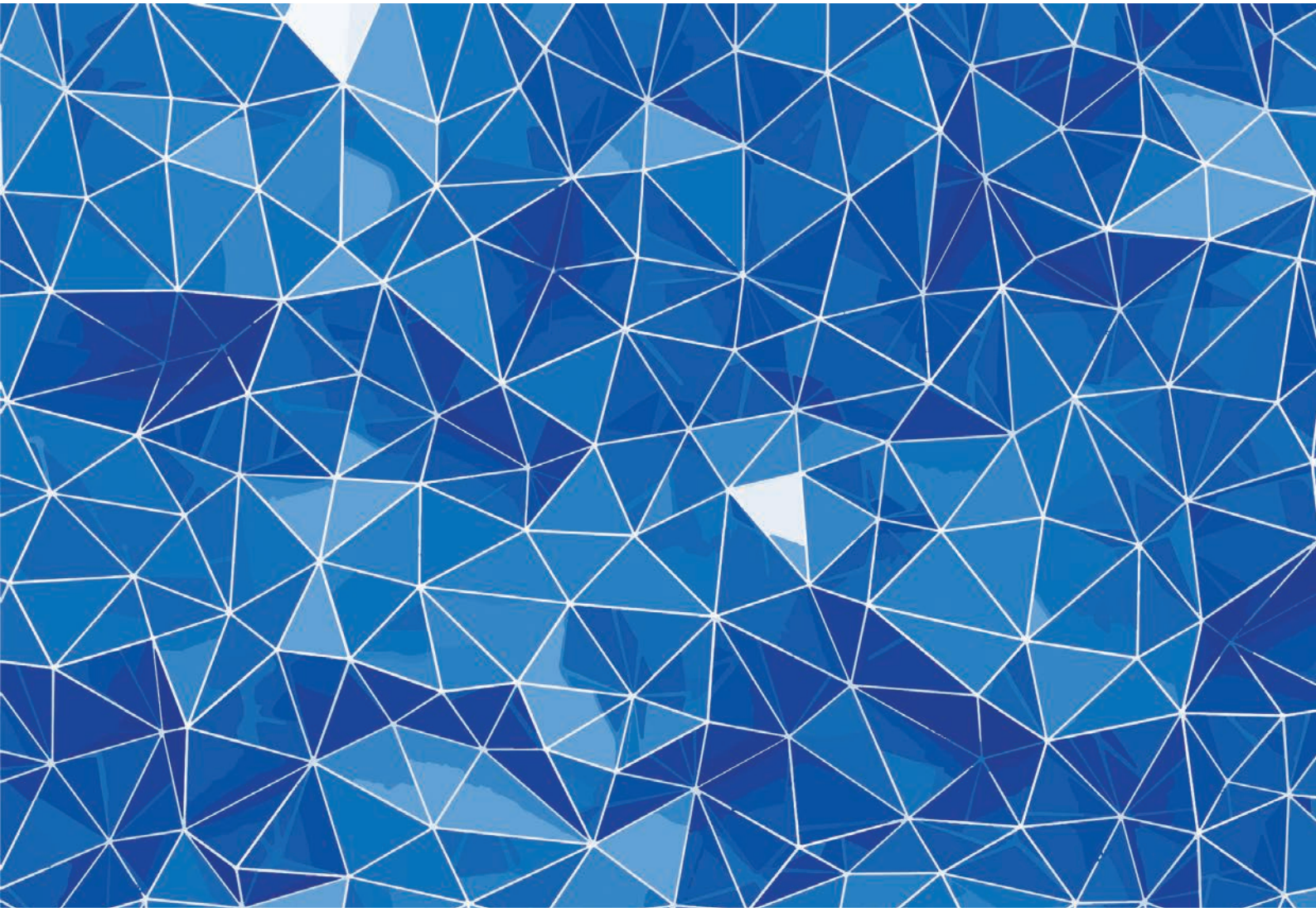
# Coding for freedom.

Mehr Berufung als Beruf:
Dein IT-Job bei der DW.

**JETZT BEWERBEN!**
**dw.com/it-karriere**

Du entwickelst passgenau? Dann fordere uns auf der **JavaLand** am Klask heraus. Du willst was footern? Komm mit uns bei einer Tüte Popcorn ins Gespräch.

Du findest uns auf der **JavaLand** direkt gegenüber vom Haupteingang.



# Native Images mit GraalVM

*Bernd Müller, Ostfalia*

*Dieser Artikel gibt einen ersten Einblick in die Erzeugung nativer Images mit der GraalVM. Neben der Möglichkeit, in verschiedenen Programmiersprachen erstellte Programmteile in ein und derselben virtuellen Maschine ausführen zu können, ist die Erzeugung nativer Images einer der Gründe, warum die GraalVM innerhalb kürzester Zeit so viel Aufmerksamkeit erlangt hat. Der Autor stellt die Grundlagen der Erzeugung nativer Images vor und zeigt, was schon geht – aber auch, was nie gehen wird. Neben diesem Artikel wird es noch weitere geben, da das Thema „native Images mit der GraalVM" sehr facettenreich ist.*

Oracle, die Firma hinter der GraalVM, beschreibt das Produkt als *„A universal virtual machine for running applications written in JavaScript, Python, Ruby, R, JVM-based languages like Java, Scala, Groovy, Kotlin, Clojure, and LLVM-based languages such as C and C++"*. Neben den schon immer in Bytecode übersetzten Sprachen Java, Scala, Groovy, Kotlin und Clojure ist die Möglichkeit einer gemeinsamen Ablaufumgebung für zusätzlich JavaScript, Python, Ruby, R sowie C und C++ das Alleinstellungsmerkmal dieser virtuellen Maschine. Für die beiden letztgenannten Sprachen gilt dies allerdings nur, falls sie nicht nativ, sondern in LLVM-Code übersetzt wurden. Die Möglichkeit des Kompilierens von Java in nativen Code ist in der Standarddistribution der GraalVM nicht enthalten, kann aber als optionale Komponente nachinstalliert werden, genauso wie die Interpreter für Python, Ruby und R.

## GraalVM: Überblick und Installation

GraalVM ist mit dem Slogan „Run Programs Faster Anywhere" *[1]* zu finden und wird als Open-Source-Projekt *[2]* entwickelt. Wenn es allerdings um den Download geht, wird zwischen Community und Enterprise Edition unterschieden. Die letztgenannte Variante ist dann nicht mehr quelloffen und muss über das Oracle Technology Network heruntergeladen werden. Für Evaluationszwecke, wie für diesen Artikel, ist die Verwendung jedoch frei möglich und wird von uns so praktiziert. Wir verwenden die aktuelle Version 19.2.1 für Linux. Die heruntergeladene Datei ist ein komprimiertes TAR, das an beliebiger Stelle entpackt wird. Wie bereits erwähnt, ist die Native-Image-Erzeugung eine optionale Komponente und muss nachinstalliert werden. Nachdem diese Komponente als JAR-Datei heruntergeladen wurde, wird sie mit dem GraalVM Component Updater (gu) installiert, und zwar mit `gu -L install <component-jar>`. Das Programm `gu` befindet sich im bin-Verzeichnis der Installation, so wie die meisten Befehle eines normalen JDK, etwa `javac`, `java`, `jar` etc. In der GraalVM Version 19.2.1 ist Java noch in der Version 8 enthalten, an der Version 11 wird gearbeitet. Nach dieser Aktualisierung steht mit `native-image` ein weiterer Befehl im bin-Verzeichnis zur Verfügung und wartet auf seinen ersten Einsatz.

## Zunächst ein wenig Java-Compiler-Geschichte

In der Java aktuell 04/2019 *[3]* haben wir Javas Just-in-time-Compiler beleuchtet. Die beiden JIT-Compiler, C1 und C2 genannt, sind etwas in die Jahre gekommen und Oracle arbeitet an einem Nachfolger mit dem Namen „Graal". Während C1 und C2 wie fast alle Module der HotSpot-Virtual-Machine in C++ geschrieben sind, ist Graal in Java implementiert. Grund hierfür waren die Erfahrungen mit C1 und C2, die praktisch nicht mehr wartbar waren und deshalb neu implementiert werden sollten. Historisch gesehen starteten die Compiler-Arbeiten im OpenJDK zunächst mit dem JEP 296: *Ahead-of-Time Compilation* *[4]* bereits im Jahr 2016. Ziel war es, Bytecode vor dem Start der JVM in Maschinencode umzuwandeln. Auf Basis dieses JEP wurde 2017 der JEP 317: *Experimental Java-Based JIT-Compiler* *[5]* formuliert. Der zu entwickelnde Compiler wurde Graal getauft. Sowohl der AoT-Compiler, der Graal als Code-Erzeugungs-Backend verwendet, als auch Graal selbst sind seit Java 9 in der Linux-Version des OpenJDK enthalten, bei Windows ab Version 10.

| Spracheigenschaft | Unterstützung |
|---|---|
| Dynamic Class Loading/Unloading | Not supported |
| Reflection | Supported (Requires Configuration) |
| Dynamic Proxy | Supported (Requires Configuration) |
| Java Native Interface | Mostly supported |
| Unsafe Memory Access | Mostly supported |
| Class Initializers | Supported |
| InvokeDynamic & Method Handles | Not supported |
| Lambda Expressions | Supported |
| Synchronized, wait and notify | Supported |
| Finalizers | Not supported |
| References | Mostly supported |
| Threads | Supported |
| Identity Hash Code | Supported |
| Security Manager | Not supported |
| JVMTI, JMX, other native VM interfaces | Not supported |
| JCA Security Services | Supported |

*Tabelle 1: Einschränkungen der Native-Image-Erzeugung [7]*

## Wie geht AoT-Kompilierung?

Der neue Befehl jaotc (Java Ahead-of-Time Compiler) im `bin`-Verzeichnis des JDK erlaubt die Kompilierung einer Klasse oder eines Moduls in nativen Bibliothekscode, also `.so` unter Linux bzw. `.dll` unter Windows. Das Kompilieren einer Klasse erfolgt mit: `jaotc --output libhelloworld.so HelloWorld`. Voraussetzung ist hier, dass der Java-Quellcode der Klasse zuvor in Bytecode übersetzt wurde. Die entstehende Datei ist bei einem normalen Hello-World-Programm etwa 420 Bytes groß. Die Option `--verbose` führt dazu, dass die kompilierten Methoden ausgegeben werden, hier der Default-Konstruktor und die `main()`-Methode.

Um das Kompilat zu verwenden, muss mit `java` die Verwendung der Bibliothek angezeigt werden: `java -XX:AOTLibrary=./libhelloworld.so HelloWorld`. Das Interessante dabei ist die interne Funktionsweise. Die Bytecode-Datei muss weiterhin vorhanden sein, da die JVM versucht, diese zu laden. Beim Laden werden dann die Methodendeskriptoren auf den Code in der Bibliothek umgelenkt. Die Angabe der Option `-XX:+PrintAOT` beim letzten Befehl macht dies explizit.

Wie oben erwähnt, können nicht nur Klassen, sondern auch Module AoT-kompiliert werden. So kann etwa das gesamte Basismodul mit `jaotc --output libjavabase.so --module java.base` kompiliert werden. Das Kompilieren ist durchaus zeitaufwendig und resultiert in einer Bibliotheksgröße von 323 MB. Durch eine Compiler-Option, die die zu kompilierenden Dateien einschränkt, kann dies reduziert werden. Die Option `jaotc ... --compile-commands compileCommands.txt` verweist auf die Datei compileCommands.txt, die lediglich `compileOnly java.lang.*` als Inhalt enthält. Das Kompilieren benötigt dann nur noch einen Bruchteil der ursprünglichen Zeit, die Bibliotheksgröße reduziert sich auf 40 MB. Beim Programmstart müssen beide Bibliotheken verwendet werden: `java -XX:AOTLibrary=./libhelloworld.so,./libjavabase.so HelloWorld`.

Nachdem das OpenJDK bereits seit Version 9 ahead-of-time kompilieren kann, muss jetzt im Hinblick auf die Erzeugung nativer Images nur noch verhindert werden, dass komplette Module – im Worst Case sogar alles, was sich im Klassenpfad befindet – übersetzt werden, um ein natives Image zu erzeugen. Kein leichtes Unterfangen, wie wir sehen werden.

## Wie funktioniert die Native-Image-Erzeugung mit der GraalVM?

Szenenwechsel: GraalVM. Beim Erzeugen eines nativen Image mit der GraalVM analysiert der Compiler die Klassen der Anwendung einschließlich aller Abhängigkeiten. Diese Analyse findet zur Compile-Zeit statt und kann daher Informationen, die die HotSpot VM nur zur Laufzeit hat, nicht verwenden. Grundlage der Analyse ist eine Closed-World-Assumption: Alles, was in der statischen Analyse erreichbar ist, wird auch verwendet, sprich kompiliert. Was nicht

erreichbar ist, wird nicht kompiliert. Falls etwa `Class.forName()` über einen Ausdruck lädt, der erst zur Laufzeit feststeht, ist die so zu ladende Klasse nicht zur Compile-Zeit bestimmbar und wird daher nicht kompiliert. Es ist in diesem Fall allerdings möglich, den Klassennamen dem `native-image`-Befehl als Parameter zu übergeben und somit auch zu kompilieren, wie wir in einem späteren Beispiel sehen. Da Java eine hochdynamische Sprache ist und viele Dinge erst zur Laufzeit feststehen, können eine ganze Reihe von Spracheigenschaften nicht statisch kompiliert werden. *Tabelle 1* gibt einen Überblick darüber, wie weit verschiedene Spracheigenschaften von Java bei der nativen Image-Erzeugung unterstützt werden.

## Was kann ein natives Image?

In *Tabelle 1* sind einige Spracheigenschaften mit *„not supported"* gekennzeichnet. Dies ist in großen Teilen nicht nur eine Momentaufnahme des aktuellen Zustands, sondern als endgültige Entwurfsentscheidung zu verstehen. Das bedeutet, dass ein Java-Programm, das mit der GraalVM in ein natives Image übersetzt wurde, nicht alle Möglichkeiten Javas unterstützt. Etwas härter ausgedrückt: Die Sprache, die durch native Images unterstützt wird, ist *nicht* Java, sondern eine echte Teilmenge von Java!

Das hört sich jetzt erst einmal sehr schlimm an, ist es aber in der Praxis nicht unbedingt, wie viele Beispiele zeigen. Quarkus, Heldion, Micronaut, Spring Boot, Spark und weitere setzen auf die GraalVM beziehungsweise überlegen, wie weit sie dabei gehen wollen. Sie müssen sich daher einschränken, dürfen also nicht den gesamten Sprachumfang von Java nutzen. Wie es aber scheint, ist dies kein K.o.-Kriterium und tut der Popularität der GraalVM keinen Abbruch. Um die notwendigen Beschränkungen besser verstehen zu können, muss man sich klarmachen, wie ein natives Image und dessen Erzeugung funktioniert. Die Definition von Java erfolgt nicht ausschließlich über die Spezifikation der Sprache *[8]*, sondern auch über die Spezifikation der virtuellen Maschine *[9]*. Existiert diese nicht, entfällt ein großer Teil der Spezifikation von Java und natürlich auch der Möglichkeiten von Java. So ist zum Beispiel das Kapitel 5 der JVM-Spezifikation *„Loading, Linking, and Initializing"* hinfällig. Bei einem nativen Image gibt es keine HotSpot VM, die diesem Kapitel fünf gehorcht, sondern die sogenannte Substrate VM. Diese realisiert grundlegende Dinge wie Speicher-Management, Thread Scheduling und Garbage Collection, nicht aber das Laden von Klassen. Die Substrate VM ist selbst in Java geschrieben und wird in das native Image kompiliert. Das besagte Kapitel fünf beschreibt, wie Bytecode in die JVM (HotSpot, nicht Substrate) geladen, gelinkt und initialisiert wird. Die Methode `loadClass()` der Klasse `ClassLoader` und deren Unterklassen erzeugen aus einem Byte-Array eine Instanz der Klasse `Class`. Damit kann eine Substrate VM nichts anfangen. Darum das *„not Supported"* in der ersten Zeile von *Tabelle 1*.

## Die ersten Gehversuche mit `native-image`

Nachdem der Autor den Leser nun eventuell etwas desillusioniert hat, beginnen wir mit den motivierenden Dingen, dem obligatorischen „Hello World". Dies kann mit `native-image HelloWorld` übersetzt werden. Es fällt auf, dass dies der Klassenname und nicht

```
public class HelloWorld {

  public static void main(String[] args) throws Exception {
    Method method = HelloWorld.class
        .getDeclaredMethod("helloWorld", new Class[] {});
        method.invoke(null, new Object[] {});
  }

  private static void helloWorld() {
    System.out.println("Hello World");
  }
}
```

*Listing 1: Hello World mit Reflection*

der Dateiname ist. Der Befehl `native-image` arbeitet auf Bytecode, nicht auf Java-Quellcode, sodass zuvor der normale Java-Compiler (`javac`) aufzurufen ist. Optional kann als weiterer Parameter der Image-Name angegeben werden, der im Default-Fall zum kleingeschriebenen Klassennamen wird. Das erzeugte Image ist unter Linux ein normales, dynamisch gelinktes ELF-Executable mit einer Größe von 2,4 MB. Für eine Quellcode-Größe von 116 Bytes (wir verzichten auf die Angabe des Hello-World-Quell-Codes) recht beachtlich. Dies relativiert sich jedoch etwas, wenn man sich klarmacht, dass die Substrate VM mit Speicher-Management, Thread Scheduling und Garbage Collection enthalten ist. Zusätzlich müssen alle Klassen, die transitiv erreichbar sind, kompiliert und gelinkt werden. Ein Hello-World verwendet auf oberster Ebene lediglich `java.lang.String` und `java.lang.System`. Der interessierte Leser kann einen Blick in die Import-Listen dieser beiden Klassen werfen, um einschätzen zu können, wie viele Imports diese beiden Klassen enthalten und wie die transitive Hülle aussehen könnte.

Mit der Option `--shared` kann eine (shared) Bibliothek, also eine `.so`-Datei erzeugt werden. Zusätzlich werden auch noch Include-Dateien für C bzw. C++ erzeugt. Wir gehen hierauf jedoch nicht ein und zeigen daher nicht, wie der mögliche Aufruf einer Java-Methode aus C bzw. C++ heraus ohne die Verwendung von JNI möglich ist.
Die Option `--static` erzeugt ein statisch gelinktes ELF-Executable mit einer Größe von 4,7 MB. Diese Form des Executable ist je nach Konfiguration des Linux-Systems zu empfehlen, da neben den typischen Verdächtigen, wie etwa `libc` und `libm`, auch `libz` verwendet wird, die eventuell nicht auf allen Systemen vorhanden ist.

## Reflektives Hello World

In *Tabelle 1* ist vermerkt, dass die Verwendung von Reflection ein gewisses Maß an Konfigurationsaufwand benötigt. Dies ist für den allgemeinen Fall korrekt, der Trivialfall funktioniert aber ohne weiteres Zutun. Der in *Listing 1* gezeigte Code bildet ein Hello World mit Reflection ab. Dieser Code kann ganz normal kompiliert und ausgeführt werden. Eine Konfiguration ist nicht vonnöten. Der Autor schließt hier seine ersten Beispiele zur Erzeugung nativer Images mit der GraalVM und verweist auf seinen nächsten Artikel in einer zukünftigen Ausgabe der Java aktuell.

## Zusammenfassung

Nach einem kurzen Überblick über die GraalVM haben wir uns mit Java-Compilern beschäftigt; dem allseits bekannten und nicht näher beleuchteten `javac`, den JIT-Compilern C1 und C2 sowie vor allem Graal. Dieser wurde als zukünftiger Ersatz für C2 entwickelt und ist im JDK unter Linux seit Version 9, unter Windows seit Version 10 verfügbar. Graal ist so flexibel einsetzbar, dass er in der GraalVM als Ahead-of-time-Compiler verwendet wird, um native Images zu erzeugen. Wir haben dann die ersten Gehversuche bei der Erzeugung nativer Images auf Hello-World-Niveau erfolgreich absolviert. In einem zukünftigen Artikel werden wir diese ersten Gehversuche hinter uns lassen und neue Möglichkeiten der GraalVM bei der Erzeugung nativer Images kennenlernen. Stay tuned!

## Referenzen


[1] *https://www.graalvm.org*
[2] *https://github.com/oracle/graal/*
[3] Bernd Müller. Unbekannte Kostbarkeiten des SDK – Heute: Just-in-Time-Compilation. Java aktuell 04/2019.
[4] JEP 296: Ahead-of-Time Compilation. *http://openjdk.java.net/jeps/295*
[5] JEP 317: Experimental Java-Based JIT Compiler. *http://openjdk.java.net/jeps/317*
[6] JEP 243: Java-Level JVM Compiler Interface. *http://openjdk.java.net/jeps/243*
[7] *https://github.com/oracle/graal/blob/master/substratevm/LIMITATIONS.md*
[8] The Java Language Specification, Java SE 13 Edition, 2019.
[9] The Java Virtual Machine Specification, Java SE 13 Edition, 2019.



**Bernd Müller**
Ostfalia
*bernd.mueller@ostfalia.de*


Nach seinem Studium der Informatik und der Promotion arbeitete Bernd Müller für die IBM und die HDI Informationssysteme. Er ist Professor, Geschäftsführer, Autor mehrerer Bücher zu den Themen JSF und JPA, sowie Speaker auf nationalen und internationalen Konferenzen. Er engagiert sich im iJUG und speziell in der JUG Ostfalen.

# HAST DU ES SCHON MIT EINEM NEUSTART VERSUCHT?

## Come and join the CI Crowd.

JAVA DEVELOPER BI BERATER UX DESIGNER MOBILE DEVELOPER DU

**Hier neu starten:**
karriereportal.
cologne-intelligence.de

COLOGNE INTELLIGENCE

QAWARE
SOFTWARE ENGINEERING


# IT-Probleme lösen.
# Digitale Zukunft gestalten.

Mit Erfindergeist
und Handwerksstolz.

Great Place To Work.
2019
Beste Arbeitgeber™
Information & Telekommunikation

kununu.com/qaware
qaware.de/karriere