# Java aktuell

iJUG Verbund
www.ijug.eu

**Build Tools**
Maven Daemon, Maven Enforcer, Gradle

**Kotlin**
Für Backend-Applikationen

**Performance**
Performance-Analyse mit Lastkurven

# BUILD TOOLS

Ausgabe 05/2023

Java aktuell

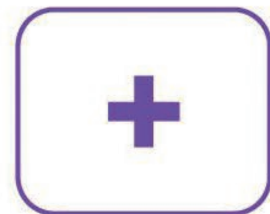
Praxis. Wissen. Networking. Das Magazin für Entwickler

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

# Eigenständige Java-Applikationen erstellen

**Christian Heitzmann, SimplexaCode AG**

*Java-Applikationen werden normalerweise direkt als Bytecode (also .class-Dateien) paketiert und verteilt. Das ist aber oft nicht wünschenswert. Zum einen lässt sich der Quellcode – und damit die Programmlogik – relativ einfach aus dem Bytecode rekonstruieren, zum anderen benötigt der Endanwender immer eine installierte Java Virtual Machine. Dieser Artikel zeigt auf, welche Möglichkeiten es heute gibt, Java-Applikationen zu erstellen, die eigenständig lauffähig sind.*



</>

JAVA

jlink

Die Motivation zu diesem Artikel hat sicher ein Stück weit mit meiner persönlichen Java-Geschichte zu tun: Vor ziemlich genau 20 Jahren habe ich in der damaligen Firma, in der ich während eines Zwischenjahres angestellt war, nach ausgiebigen Recherchen den „kühnen" Vorschlag unterbreitet, für künftige Lösungen doch besser auf Java zu setzen (damals noch in der Version 1.4). Im Bereich der Verkehrstechnik wurden in neuen öffentlichen Ausschreibungen nämlich zunehmend Internettechnologien gefordert. Für ein Unternehmen, das bislang seine Verkehrsrechner auf „Bit- und Byte-Ebene" in Assembler und in proprietären Intel-Hochsprachen programmiert hat, war das natürlich absolutes Neuland.

Schnell musste der Vergleich her, ob Java-Programme gleich gut performen wie C/C++-Programme – eine unnötige Diskussion, die mich in meiner Laufbahn noch weitere fünf Jahre verfolgen sollte, heute aber Gott sei Dank obsolet geworden ist. Die Gründe dafür erfahren Sie im Laufe dieses Artikels.

Ebenfalls war der Wunsch, Java-Programme gleich „bequem" verpacken zu können, wie es C/C++-Compiler tun – im Idealfall in Form einer einzigen `.exe`-Datei. Dieses Anliegen war durchaus berechtigt, nicht nur in der damaligen Firma, sondern auch in vielen anschließenden Projekten, in denen ich mir so etwas gewünscht hätte. Umso überraschender ist es, dass es weitere 15 Jahre dauerte, ehe sich Oracle dieses Anliegens offiziell angenommen hat.

## Geschichte

In den Jahren 2003 und 2004 habe ich mich also schon intensiv mit der Erstellung eigenständiger Java-Applikationen beschäftigt.

Marktführer dazumal war ganz klar Excelsior JET, ein Native Compiler, der aus Java-(SE-)Code direkt ausführbare Programme für Windows, Mac und Linux generieren konnte *[1]*. 2018 besaß ich noch eine Lizenz, ehe 2019 die russische Firma aus Nowosibirsk nach 20 Jahren ganz überraschend und kurzfristig ihre Tätigkeit einstellte. Über die Gründe wurde offiziell nie etwas bekannt. Es kann nur spekuliert werden, dass die GraalVM – um die es nachher in diesem Artikel natürlich auch geht – deren Geschäft den Todesstoß verpasst hat.

Als freien nativen Compiler gab es den GNU Compiler for Java (GCJ) *[1]*, der nach meiner Erinnerung aber nie richtig funktioniert hat, was vorrangig seiner unvollständigen (eigenen!) Implementierung der Standardklassenbibliotheken (GNU Classpath) geschuldet war. 2016 wurde der GCJ komplett eingestellt.

Persönlich sehr überzeugt hat mich dazumal hingegen Borland JBuilder X, der in der Developer- und Enterprise-Variante über einen Wrapper verfügte, mit dem sich Java-Applikationen in Windows, Mac und Linux (und sogar Solaris) via Doppelklick auf eine einzige Executable-Datei ausführen ließen. Das hat auch immer sehr gut funktioniert. Etwa im gleichen Zeitraum wurde das freie Eclipse zunehmend populärer, das aber leider nie ein derartiges Feature bot. Spätere JBuilder-Versionen basierten dann auf Eclipse, und Borland wurde von Embarcadero Technologies aufgekauft. Die letzte JBuilder-Version erschien 2009.

Der Fairness halber muss erwähnt werden, dass der JBuilder-Wrapper lediglich ein kleines Executable enthielt, das den `java`-Aufruf für das jeweilige Betriebssystem tätigte. Java musste beim Anwender also bereits installiert sein. Der Wrapper hat dann die in seinem Executable-Archiv enthaltenen `.class`-Dateien temporär im Arbeitsspeicher extrahiert und der JVM beim Aufruf mitgegeben. Wahrlich keine Hexerei, da sich eine solche Lösung mit überschaubarem Aufwand auch selber realisieren ließe, aber durchaus bequem, stabil und damals seiner Zeit voraus.

## Eigenständige Java-Applikationen

Unter eigenständig lauffähigen Java-Applikationen soll verstanden werden, dass der (End-)Benutzer keine eigene Java Virtual Machine installiert haben muss, um sie auszuführen.

An sich mag eine solche Installation kein großes Problem (gewesen) sein. In den ersten Java-Versionen (1.4 bis 8) war es trivial, wenn auch wegen des einnistenden Java Web Start und ständiger Hintergrund-Updates meines Erachtens nie ganz nebenwirkungsfrei. Ebenfalls haftete Java dazumal immer noch leicht der Ruf eines „Sicherheitsrisikos" an, was wohl den früheren Java Applets geschuldet war (die aber heute zum Glück alle aus dem Verkehr gezogen sind). Mit der Umstellung des Oracle-Lizenz- und Versionierungsmodells ab Java 9, bei der technisch wie politisch kein Stein mehr auf dem anderen blieb, wurde das Auffinden und Installieren eines korrekten JDKs geradezu lästig. Mittlerweile scheint sich das mit einem halben Dutzend unkomplizierter OpenJDK-Builds, wie zum Beispiel Adoptium Eclipse Temurin (früher AdoptOpenJDK), und dem sich eingespielten Releasesystem wieder gelegt zu haben.

Das ändert aber nichts daran, das ausgelieferte Java-Applikationen oftmals auf eine ganz bestimmte JDK-Version setzen. Die Abwärtskompatibilität von Java funktionierte zwar rückblickend betrachtet immer ganz gut, aber nicht perfekt. Das mag wohl der Grund sein, wieso jede nennenswerte Java-Desktop-Applikation, die ich kenne, praktisch immer auch ihre eigene JDK-Runtime mitliefert.

Nicht Teil des vorliegenden Artikels soll die gesamte (Native-)Cloud-Thematik sein. Es leuchtet ein, dass sich die Anforderungen an eine Microservice-Architektur, namentlich minimaler Speicherplatzbedarf, hochoptimierter Code sowie schnelles Aufstarten und Herunterfahren, bevorzugt mit nativ vorkompilierten Java-Applikationen umsetzen lassen *[2]*.

## Application Images mit jlink

Mit Java 9 wurde das gesamte JDK modularisiert. Gleichzeitig erhielt das Kommandozeilentool `jlink` *[3][4][5]* Einzug in ebendieses. Damit ist es möglich, ein eigenes Application Image zu erstellen (in der Fachsprache: zu linken), das *nur* die für die Ausführung notwendigen Module des JDKs enthält. Eine typische JDK-Installation umfasst heute zwischen 300 und 400 MB. Die meisten Module, wie AWT, Swing oder CORBA, werden nicht benötigt, sodass es mehr als nachvollziehbar ist, wenn zur Ausführung beim Endbenutzer nicht jeweils der ganze Koloss mitgeliefert werden soll. Dank des Modulsystems sind ja alle (transitiven) Abhängigkeiten klar definiert. *Listing 1* zeigt einen typischen `jlink`-Aufruf.

- Mit `--module-path` werden alle Modulpfade angegeben; hier die Module des Eclipse Temurin Java-19-JDKs sowie – mit `:` getrennt (unter Windows mit `;`) – die eigenen Applikationsmodule,

```
jlink --module-path ~/Library/Java/JavaVirtualMachines/temurin-19/Contents/Home/jmods/:./out/production/
      --add-modules slideshow
      --launcher play=slideshow/ch.simplexacode.slideshow.SlideShowMain
      --output slideshow_executable
```

*Listing 1*

die sich hier im Beispiel im Ausgabeverzeichnis `out/production` befinden.

- `--add-modules` gibt die eigentlichen Namen der einzubindenen Applikationsmodule an. In diesem Beispiel handelt es sich einzig um das Modul `slideshow`.
- Mit `--launcher` wird der Linker angewiesen, einen expliziten Launcher anzulegen, also eine ausführbare Datei, die sich einfach auf der Kommandozeile oder mit Doppelklick ausführen lässt. In diesem Beispiel soll dieser Launcher `play` heißen und auf die Main-Klasse `SlideShowMain` im Package `ch.simplexacode.slideshow` des Moduls `slideshow` verweisen. Fehlt ein solcher Launcher, muss die Applikation inkl. aller Pfade separat auf Kommandozeile mit `java` gestartet werden, womit sich das Pfad-Gepfriemel zum Aufrufer verlagert.
- `--output` gibt den Namen des Verzeichnisses an, in dem das Application Image erzeugt wird; hier `slideshow_executable`.

Das Artefakt von `jlink` ist ein Verzeichnis, bestehend aus vielen Dutzend Dateien und diversen Unterverzeichnissen. Wurde ein Launcher erzeugt, so befindet sich dieser im Unterverzeichnis `bin/`. Die Applikation lässt sich also via Doppelklick auf beziehungsweise Kommandozeilenaufruf von `slideshow_executable/bin/play` starten, selbst wenn auf dem ausführenden Rechner kein JDK installiert ist. Das JDK ist ja Teil des Application Images (oder vielmehr des Application Directories).

Mit den Optionen `--compress=2`, `--strip-debug`, `--no-header-files` und `--no-man-pages` lassen sich bezüglich des Speicherplatzbedarfs noch deutliche Optimierungen erzielen. 40 % Einsparung sind je nach Anwendung problemlos möglich.

`jlink` ermöglicht es auch, Application Images für andere Betriebssysteme als das eigene zu erzeugen. Letztlich muss hierzu nur das JDK des Zielsystems heruntergeladen und im `--module-path` eingebunden werden. `jlink` erkennt, dass es sich um das JDK eines anderen Betriebssystems handelt, und erstellt die dazugehörigen Binärdateien *[6]*.

Für weitergehende Informationen sei vor allem an *[4]*, *[6]* und die vielen einfach auffindbaren Online-Quellen verwiesen.

## Installationspakete mit jpackage

Unsere Anforderung, beim Anwender kein separat installiertes JDK zu verlangen, können wir jetzt mit `jlink` erfüllen. Nicht schön hingegen ist das Deployment unserer Applikation als ganze Verzeichnisstruktur statt einer einzigen Datei.

Mit `jpackage` steht seit Java 16 ein Kommandozeilentool zur Verfügung, das Installationspakete in den üblichen Formaten für die Betriebssysteme Windows, Mac und Linux erstellt *[7]*. Für Windows sind das wahlweise `.exe` oder `.msi`, für Mac `.pkg` oder `.app` innerhalb eines `.dmg`, und für Linux `.deb` oder `.rpm`. Ein „Cross-Packaging“ ist aber nicht möglich, das heißt, `jpackage` muss – im Gegensatz zu `jlink` – immer auf dem Zielbetriebssystem ausgeführt werden.

Ob die so deployten Applikationen beim Anwender wirklich *installiert* werden müssen, ist nicht sicher. Zumindest unter Mac (meinem Heimsystem) kann ich die so erzeugten Applikationen auch direkt aus dem `.dmg`-Image starten. Vorsicht ist geboten, wenn die Anwendung Konsolenausgaben produziert oder Aufrufparameter erfordert. Zumindest auf Mac ist ein derartiger „Low-Level-Zugriff“ deutlich harziger als der direkte Umgang mit der Launcher-Binärdatei, wie sie `jlink` aus dem vorherigen Abschnitt erzeugt.

Der Aufruf von `jpackage` mit Standardoptionen ist relativ unspektakulär und selbsterklärend, wie *Listing 2* an unserer bereits bekannten hypothetischen Slideshow-Applikation zeigt. Für weitere Details sei auf *[7]* verwiesen, das auch die verschiedenen Varianten bei modularer und nicht modularer Applikation sowie gegebenen `.jar`-, `.jmod`- und reinen Source-Dateien behandelt.

## Native Images mit GraalVM

Der „heilige Gral“ in Sachen eigenständiger Java-Applikationen ist natürlich die neue GraalVM. Diese muss zuerst separat von *[8]* heruntergeladen und installiert werden. Es gibt sie in einer kostenlosen Community und einer kostenpflichtigen Enterprise Edition. Letztere soll für Java zirka 20 % und für dynamische Sprachen etwa 50 % schneller sein, einen nur halb so großen Speicherbedarf aufweisen und diverse Sicherheitsmerkmale haben *[2]*. Am interessantesten dürfte aber die Profile Guided Optimization (PGO) sein, auf die im Abschnitt *Performanzaspekte* kurz eingegangen wird.

Nach der Installation der eigentlichen GraalVM muss mittels `gu install native-image` noch die Erweiterung für die Erzeugung nativ kompilierter Images hinzugefügt werden. `gu` ist der hauseigene **G**raalVM **U**pdater. Am Schluss sollte auch der Kommandozeilenbefehl `native-image` zur Verfügung stehen.

Der Aufruf von `native-image` erfolgt analog einem Aufruf von `java`

```
jpackage --module-path ~/Library/Java/JavaVirtualMachines/temurin-19/Contents/Home/jmods/:./out/production/
         --module slideshow/ch.simplexacode.slideshow.SlideShowMain
         --name slideshow_installer
         --app-version 2023.06
```

*Listing 2*

[9]. Bleiben wir bei unserer SlideShow-Applikation, dann können wir mit dem Befehl wie in *Listing 3* gezeigt ein natives Kompilat erzeugen.

- In `--module-path` genügt es, nur noch auf die Module der eigenen Applikation zu verweisen. Die GraalVM enthält quasi ihr eigenes JDK, das implizit zur Verfügung steht. Allerdings arbeitet GraalVM (Stand: Juni 2023) noch mit Java 17 und akzeptiert keine Applikationsklassen, deren Bytecodes für eine neuere Java-Version kompiliert wurden.
- `-m` gibt die Main-Klasse an.
- `-o` bestimmt den Namen der einen und einzigen ausführbaren Datei.

Seien Sie nicht überrascht, wenn der Kompiliervorgang sehr lange dauert. Eine überschaubare Swing-Applikation hat bei meinen Tests bereits 30 Sekunden Zeit benötigt.

Stichworte Swing und AWT: Hier offenbart sich bereits ein erster Bug. Eine AWT- (und damit auch Swing-)Applikation stürzt beim Aufruf direkt ab, was offenbar mit den Schriften und wohl auch etwas mit Mac zu tun hat *[10][11]*.

Über die GraalVM und ihre Native Images wurde schon viel geschrieben, vor allem hier in Java aktuell *[12][13][14]*, aber auch in JavaSPEKTRUM *[15][16][17]*. Mit *[2]* ist über sie sogar ein ganzes Buch erschienen. Von daher nur ganz kurz die wichtigsten Punkte:

GraalVM ist – anders als ihr Name vermuten lässt – nicht nur eine Virtual Machine, sondern auch ein Compiler (Graal Compiler), der nebst Java nicht nur andere JVM-Sprachen, sondern (via Truffle) sogar ganz andere Programmiersprachen wie Python oder JavaScript unterstützt. Genau genommen kommt Graal nicht nur mit einer, sondern sogar mit zwei virtuellen Maschinen daher: die zweite heißt SubstrateVM.

Der Graal Compiler kann eine sogenannte Ahead-of-Time-Kompilierung (AOT) durchführen. Dazu analysiert er ausgehend von der `main`-Methode alle erreichbaren Codestellen der eigenen Applikation, des JDKs und aller Drittmodule, und entfernt alles, was nicht aufrufbar ist (Dead Code Elimination). Den Rest übersetzt er inklusive SubstrateVM in Maschinensprache. Es ist ebenso möglich, Static Native und Shared Libraries zu bauen.

Mit der statischen Kompilierung kommt es zu einigen Einschränkungen des an sich hochdynamischen Javas. Es ist theoretisch nicht möglich, Klassen dynamisch nachzuladen, was vor allem einen Einfluss auf Spracheigenschaften wie Reflection, Classloader und dynamische Proxies hat – und damit verbunden auch auf sie basierende Frameworks wie zum Beispiel Spring (Boot) (wobei mit Spring Framework 6 neu auch Native Images unterstützt werden *[18]*). Mehr zu den Hintergründen ist unter anderem in *[13]* zu finden.

## Decompiler und Obfuscator

Javas `.class`-Dateien lassen sich prinzipiell sehr einfach dekompilieren. Nicht selten lässt sich daraus der gesamte Quellcode rekonstruieren *[19]*. Wer also auf den Schutz seines geistigen Eigentums bedacht ist, sollte unbedingt Vorkehrungen treffen.

Die Artefakte aus `jlink` und `jpackage` (das ja auf `jlink` basiert) enthalten eine große Datei namens `modules`, in der die Klassendateien der Module im JIMAGE-Format gespeichert sind *[5][20]*. Diese lassen sich einfach extrahieren und bieten daher *keinen* Schutz vor Dekompilierung.

Native Kompilate aus der GraalVM sehen da schon besser aus *[21]*. Dekompilierung ist immer ein Abwägen zwischen Aufwand und Nutzen, und das kippt bei ausführbaren Binärdateien ganz klar auf die Seite des Aufwands.

Mit einem sogenannten Obfuscator lässt sich der Java- und damit der Byte-Code „verschleiern“. Ein Obfuscator benennt z. B. alle (nicht öffentlichen) Klassen, Methoden, Variablen etc. in mühsame Namen wie `C1`, `C2`, `C3`, `m1`, `m2`, `m3`, `v1`, `v2`, `v3` etc. um, entfernt sämtliche Debug-Informationen und verkompliziert teilweise einfache Sachen wie String-Konstanten durch sie generierende Methodenaufrufe *[19]*. Der wohl bekannteste Obfuscator ist ProGuard *[22]*.

## Performanzaspekte

Die Java Virtual Machine hat sich über die letzten 28 Jahre als gelungener Schachzug herausgestellt und mehr als bewährt *[23]*. Obwohl der Code „halb“ interpretiert wird (via Zwischenstufe Bytecode), haben Compiler-Konzepte wie Just-in-time (JIT) oder HotSpot für einen Geschwindigkeitszuwachs gesorgt, der denen von statisch kompilierten Sprachen streckenweise überlegen ist. Die dynamische JVM kann nämlich während der Laufzeit eines Programmes erkennen, welche Teile des Programms besonders oft durchlaufen werden, und diese Teile („Ausführungspfade“) dann besonders hoch optimieren. Außerdem kennt die JVM die Hardware, auf der sie läuft, und kann so eine Just-in-Time-Kompilierung genau auf diese Hardware zugeschnitten durchführen, zum Beispiel, um so von diversen Befehlssatzerweiterungen zu profitieren *[2][24]*.

Solche dynamischen Optimierungen können die statischen Kompilate der GraalVM nicht vornehmen. Die Meinungen gehen auseinander, ob Native Images schneller oder langsamer sind *[2]*. Die wahrscheinlich einzig korrekte Antwort darauf ist: „Es kommt darauf an.“ Wem Performanz wirklich sehr wichtig ist (Ich darf an den Spruch von Donald Knuth erinnern: „Premature optimization is the root of all evil.“), der sei an meinen Fachartikel *[25]* zu den Grundzügen der Laufzeitmessung in Java verwiesen.

Die GraalVM *Enterprise Edition* bietet eine sogenannte Profile-Guided Optimization (PGO) an. Dort wird zuerst ein instrumentiertes AOT-Kompilat erzeugt, das Messroutinen für das Ausführungsverhalten

```
native-image --module-path ./out/production/
             -m slideshow/ch.simplexacode.slideshow.SlideShowMain
             -o slideshow_native
```

*Listing 3*

enthält. Anschließend wird die Anwendung (eventuell für längere Zeit) laufen gelassen. Abschließend wird erneut ein AOT-Kompilat erzeugt, diesmal unter Einbezug der vorher ermittelten Profildaten, sodass der Graal-Compiler letztendlich ein hochoptimiertes Native Image erstellen kann *[2][9]*.

## Fazit

Es gibt heute mehrere Möglichkeiten, eigenständige Java-Applikationen zu erstellen: mittels `jlink`, `jpackage` oder *GraalVM*. Alle Lösungen haben gemeinsam, dass sie ihre eigene Ausführungsumgebung mitbringen, und der Benutzer so keine separate JVM installieren muss. Unterschiede gibt es bezüglich des Deployment-Komforts (von ganzen Verzeichnisbäumen über Launcher und Installer bis hin zu einer einzigen Executable-Datei), aber auch hinsichtlich des Schutzes vor Dekompilierung des darunterliegenden (Byte-)Codes. Performanz sollte kein ausschlaggebendes Kriterium sein, sondern bedarf im Einzelfall einer genauen, systematischen Untersuchung. Ebenfalls sind die benutzten Frameworks zu berücksichtigen, und mit welchen Kompiliermöglichkeiten sich diese vertragen. Nicht ohne Grund sind ja neue Frameworks wie Quarkus entstanden, die dem Native-Ansatz vor allem auf der Cloud besser gerecht werden.

## Referenzen


[1] C. Ullenboom, Java ist auch eine Insel, 10. Auflage, Galileo Computing, 2010

[2] A B V. Kumar, Supercharge Your Applications with GraalVM, Packt Publishing, 2021

[3] M. Inden, Java – Die Neuerungen in Version 9 bis 14, dpunkt.verlag, 2020

[4] M. Hunger, Der Java-Linker jlink, in: JavaSPEKTRUM, 2/2022, SIGS DATACOM, 2022

[5] Oracle University, Java SE: Exploiting Modularity and Other New Features, Oracle, 2018

[6] Dev.Java, Creating Runtime and Application Images with JLink, *https://dev.java/learn/jlink/*

[7] Packaging Tool User's Guide, *https://docs.oracle.com/en/java/javase/20/jpackage/*

[8] GraalVM, *https://www.graalvm.org*

[9] GraalVM Native Image Quick Reference v2, *https://www.graalvm.org/latest/docs/quick-references/*

[10] GitHub, Using Java AWT Fonts gives unsatisfiedlinkerror while running native image, *https://github.com/oracle/graal/issues/2729*

[11] GitHub, no awt in java.library.path, *https://github.com/oracle/graal/issues/2842*

[12] K. Vardanyan, Eine JVM für die Cloud: die GraalVM, in: Java aktuell, 1/2020, DOAG, 2020

[13] B. Müller, Native Images mit GraalVM, in: Java aktuell, 2/2020, DOAG, 2020

[14] B. Müller, Native Images mit GraalVM: Reflection, in: Java aktuell, 1/2021, DOAG, 2021

[15] M. Hunger, Polyglot Operations in Java mit GraalVM, in: JavaSPEKTRUM, 1/2019, SIGS DATACOM, 2019

[16] M. Hunger, Vorcompilierte Programme mit GraalVM AOT, in: JavaSPEKTRUM, 2/2019, SIGS DATACOM, 2019

[17] M. Vitz, Cloud native Java-Anwendungen mit Quarkus, in: JavaSPEKTRUM, 3/2019, SIGS DATACOM, 2019

[18] M. Simons, Native Image in Spring Framework 6, in: iX, 12/2022, Heise Medien, 2022

[19] C. Ullenboom, Java SE 9 Standard-Bibliothek, 3. Auflage, Rheinwerk Verlag, 2018

[20] M. Debnath, How Modules Are Packaged in Java 9, *https://www.developer.com/design/how-modules-are-packaged-in-java-9/*

[21] GitHub, Is it possible to decompile the executable and see the original code?, *https://github.com/oracle/graal/issues/4003*

[22] GitHub, Guardsquare ProGuard, *https://github.com/Guardsquare/proguard*

[23] M. Stal, Java rocks! – Eine Liebeserklärung, in: JavaSPEKTRUM, 4/2020, SIGS DATACOM, 2020

[24] C. Ullenboom, Java ist auch eine Insel, 16. Auflage, Rheinwerk Verlag, 2022

[25] C. Heitzmann, Performanzanalysen in Java – Teil 1: Java Microbenchmarks, in: JavaSPEKTRUM, 4/2020, SIGS DATACOM, 2020, *https://link.simplexacode.ch/cj8e*

**Christian Heitzmann**

*christian.heitzmann@simplexacode.ch*


Christian Heitzmann ist Java-, Python- und Spring-zertifizierter Softwareentwickler mit einem CAS in Machine Learning und Inhaber der SimplexaCode AG in Luzern. Er entwickelt seit über 20 Jahren Software und gibt seit über 12 Jahren Unterricht und Kurse im Bereich der Java- und Python-Programmierung, Mathematik und Algorithmik. Als Technical Writer dokumentiert er Softwarearchitekturen für Unternehmen.