Java aktuell

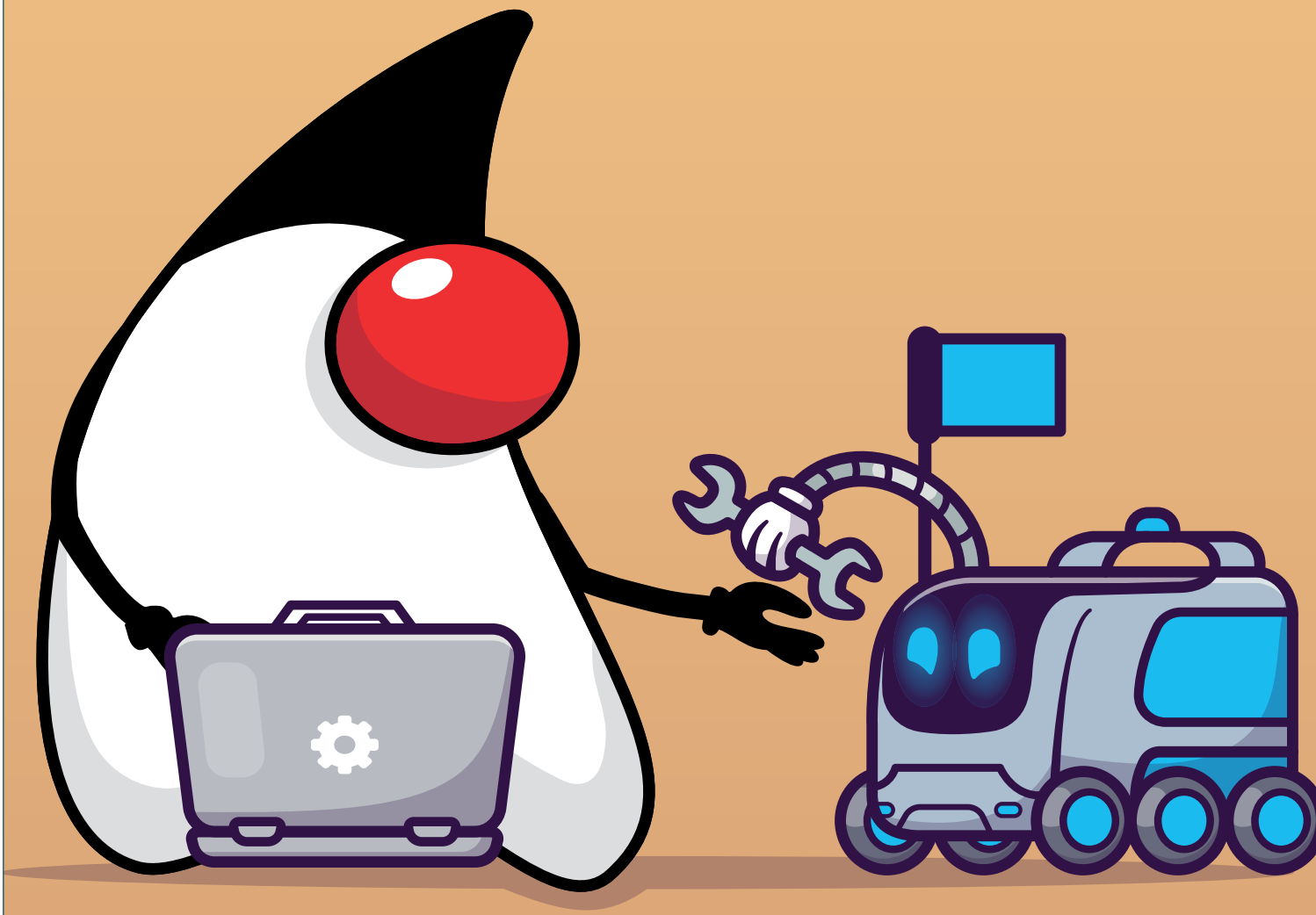
iJUG Verbund
www.ijug.eu

Tools
Shell, GitHub Actions mit Dependabot

Testing
Playwright Java, Test-First-Strategie

Cloud Native
Cloud-Native-Applikationen mit Spring Boot 3 und GraalVM

TOOLS



Ausgabe 03/2024
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 03

Invoke Dynamic – ursprünglich gedacht für dynamisch typisierte JVM-Sprachen, ist dieser Bytecode-Befehl mittlerweile das Universalwerkzeug für viele Java-Neuerungen

Bernd Müller, Ostfalia

Der Bytecode-Befehl ***`invokedynamic`*** *wurde mit Java 7 als neuer Befehl in die JVM-Spezifikation aufgenommen. Der zugrundeliegende JSR beschreibt ein damaliges wachsendes Interesse an der Ausführung verschiedener, meist dynamisch typisierter Programmiersprachen auf der JVM. Diese Ausführung sollte effizienter als mit den zu dieser Zeit vorhandenen Maschinenbefehlen möglich sein. Der Befehl wurde im JDK für Java 7 tatsächlich selbst nicht verwendet, sondern war ein Angebot an andere Programmiersprachen, die die JVM als Zielplattform hatten. Mittlerweile wird* ***`invokedynamic`*** *aber zur Realisierung sehr vieler Java-Sprach-Features eingesetzt. Von Lambda-Ausdrücken über String-Konkatenationen bis hin zu Records wird mittlerweile* ***`invokedynamic`*** *verwendet. Dieser Artikel beleuchtet die ursprüngliche Motivation des Befehls, die Attraktivität für die heutige Verwendung sowie die nach unserem Wissen aktuell tatsächlichen Verwendungen.*

Was ist invokedynamic und welche Probleme werden damit gelöst?

Java ist eine streng typisierte Sprache. Für jeden Ausdruck ist daher bereits zur Compile-Zeit der Typ, zumindest der Obertyp, bekannt. Bei dynamisch oder schwach (*weak*) typisierten Sprachen ist dies nicht so. Erst zur Laufzeit ist der Typ eines Ausdrucks bekannt und führt damit erst während der Laufzeit zur Bestimmung und Verwendung der dem Typ entsprechenden Operationen. Bei der folgenden Python-Funktion zur Addition werden keine konkreten Typinformationen verwendet:

```
def add(a,b):
     return a + b;
```

Zur Laufzeit können mit dieser Funktion sowohl ganze als auch gebrochene Zahlen addiert werden. Bei einer Laufzeitimplementierung von *Python* durch die JVM muss daher zunächst der Typ der Parameter festgestellt werden, um den korrekten JVM-Befehl aufrufen zu können. Die JVM unterstützt lediglich die Addition von Integer-, Long-, Float- und Double-Werten mit den Maschinenbefehlen `iadd`, `ladd`, `fadd` und `dadd`, aber keinen ungetypten Additionsbefehl.

Da die JVM eine sehr stabile und performante Laufzeitumgebung ist, wird sie von vielen Sprachimplementierungen gerne genutzt. Der Wikipedia-Artikel *[1]* führt hierzu eine ganz Reihe bekannter Sprachen auf. Um die Popularität der Java-Plattform, insbesondere der JVM, weiter zu erhöhen, wurde bereits 2006 der JSR 292 *[2]* ins Leben gerufen, der die Unterstützung dynamisch typisierter Sprachen, insbesondere deren performante und qualitativ hochwertige Umsetzung auf der JVM, zum Ziel hatte. Dieser JSR wurde 2011 mit der Einführung des neuen JVM-Befehls `invokedynamic`, umgangssprachlich auch häufig *indy* genannt, mit dem Release von Java 7 finalisiert. Da `invokedynamic` für Nicht-Java-Sprachen gedacht war, wurde der Befehl vom Java-7-Compiler tatsächlich nicht bei der Erzeugung von Bytecode verwendet *[3]*.

Wie funktioniert nun aber `invokedynamic`? Um dies zu klären, ist ein Blick auf die bisherigen Alternativen für Aufrufe hilfreich. Es gab vier verschiedene Möglichkeiten, Methoden aufzurufen: `invokestatic` für den Aufruf statischer Methoden, `invokeinterface` für den Aufruf von Interface-Methoden, `invokespecial` für den Aufruf von Konstruktoren, von `super()`-Konstrukten und von private-Methoden, sowie `invokevirtual` für Objekt-Methoden.

Diese vier Aufrufarten haben eines gemeinsam: Sie sind sehr eng an die Semantik der Sprache Java gebunden und damit in ihrer Umsetzung sehr stringent festgelegt, so dass sie bereits zur Compile-Zeit vollumfänglich, ihrem jeweiligen Verwendungszweck entsprechend, erzeugt werden können. Sie erlauben somit keinerlei Dynamik zur Laufzeit.

Demgegenüber erlaubt `invokedynamic` ein Bootstrap-Verfahren, um zur Laufzeit beliebigen Bytecode aufrufen zu können. Dazu liefert der Aufruf der Bootstrap-Methode (kurz BSM) eine Instanz der Klasse `CallSite` (Package `java.lang.invoke`) zurück, die ein `MethodHandle` (ebenfalls Package `java.lang.invoke`) enthält, das letztendlich aufgerufen wird. Mit dieser zusätzlichen Indirektion wird die oben angesprochene Dynamik beziehungsweise Flexibilität gewonnen, da die entsprechenden Methoden in jedem neuen Java-Release überarbeitet und verbessert werden können. Diese Flexibilität wird jedoch mit einem gewissen Mehraufwand bezahlt. Dieser muss in der Regel jedoch nur beim ersten Aufruf erbracht werden. Das in der `CallSite` zurückgelieferte `MethodHandle` wird gecacht und beim zweiten sowie weiteren Aufrufen direkt und ohne das Bootstrap-Verfahren verwendet. Das hier etwas simplifiziert dargestellte Verfahren wird im *JavaDoc* des Package `java.lang.invoke` ausführlich dargestellt.

Wir gehen im Folgenden auf einige der Verwendungen des `invokedynamic`-Befehls ein. Da uns selbst Bytecode eher fremd ist, bleiben wir auf der Ebene der Java-Bezeichner beziehungsweise generierter Bezeichner. Der geneigte Leser kann sich aber mit dem `javap`-Befehl und der Option `-v` beziehungsweise `-verbose` recht einfach davon überzeugen, dass die genannten Java-Methoden im Bytecode tatsächlich verwendet werden.

Lambda-Ausdrücke

Wir beginnen mit Lambda-Ausdrücken, die mit Java 8 eingeführt wurden und damit das erste Java-Sprachkonzept waren, das mithilfe des in Java 7 eingeführten `invokedynamic`-Befehls umgesetzt wurde. Ben Evans und Martijn Verburg erwähnen in *The Well-Grounded Java Developer [3]*, dass eine erste prototypische Implementierung von Lambda-Ausdrücken in Java 8 durch anonyme Klassen erfolgt sei, was recht nahe liegt. Diese Implementierungsalternative hatte aber eine ganz Reihe negativer Eigenschaften. So werden eine nicht unerhebliche Anzahl zusätzlicher Klassen erzeugt, die jeweils in die JVM geladen, validiert und gelinkt werden müssen. Zudem steht die Implementierung fest, da der Bytecode erzeugt und unveränderlich ist. Die beschriebene Alternative über ein Bootstrap-Verfahren erlaubt es jedoch einer späteren Implementierung, ein effizienteres Verfahren im Bootstrap-Mechanismus und der Call-Site einzusetzen und so die Performanz zu erhöhen.

Brian Goetz selbst beschreibt in *Translation of Lambda Expressions [4]* das Verfahren, um Lambda-Ausdrücke in Bytecode zu übersetzen. Die verwendete Bootstrap-Methode ist die Methode `metafactory()` der Klasse `LambdaMetafactory` im Package `java.lang.invoke`. Die ersten Zeilen des *JavaDoc* der Klasse erläutern den Einsatzzweck:

> *„Methods to facilitate the creation of simple function objects that implement one or more interfaces by delegation to a provided MethodHandle, possibly after type adaptation and partial evaluation of arguments. These methods are typically used as bootstrap methods for invokedynamic call sites, to support the lambda expression and method reference expression features of the Java Programming Language."*

Neben Lambda-Ausdrücken werden auch Methodenreferenzen mit diesem Ansatz übersetzt. Der Rückgabetyp der Methode `metafactory()` ist `CallSite`. Diese Datenstruktur einhält ein `MethodHandle`, das durch den `invokedynamic`-Befehl aufgerufen wird. Als Ziel des Method-Handles werden Methoden nach den Namensschemata `lambda$classname$0`, `lambda$classname$1` und weiteren erzeugt, die die im Quellcode verwendeten Lambda-Aus-

drücke repräsentieren. Auch hiervon kann man sich leicht mit dem `javap`-Befehl überzeugen.

String-Konkatenation

Scheinbar waren die Erfahrungen bei der Verwendung von `invokedynamic` mit Lambda-Ausdrücken in Java 8 so positiv, dass mit Java 9 ein weiteres Java-Sprach-Feature damit realisiert wurde: die Konkatenation von Strings. Wir sind auf den zugrundeliegenden JEP 280 bereits in der Ausgabe 03/2023 der *Java aktuell* [5] kurz eingegangen. Der JEP 280 [6] trägt den Titel *Indify String Concatenation* und definiert sich in der Zusammenfassung als:

> *„Change the static String-concatenation bytecode sequence generated by javac to use invokedynamic calls to JDK library functions. This will enable future optimizations of String concatenation without requiring further changes to the bytecode emitted by javac."*

Während die Konkatenation von String-Literalen mit dem Plus-Operator bereits zur Compile-Zeit erfolgt, wurden bis Java 9 String-Konkatenationen mit Variablenbeteiligung durch `StringBuilder#append()`-Aufrufe im Bytecode realisiert. Der JEP 280 hatte das Ziel, die Konkatenation von Strings in späteren Versionen der JVM performanter durchführen zu können, als mit direkt im Bytecode festgehaltenen `append()`-Aufrufen. Die Indirektion von `invokedynamic` über eine Bootstrap-Methode, die in einer späteren JVM eventuell optimiert wurde, macht dies möglich.

Die tatsächlich verwendete Bootstrap-Methode ist `StringConcatFactory#makeConcatWithConstants()`. Die Klasse befindet sich ebenfalls im Package `java.lang.invoke` und die ersten Zeilen des *JavaDocs* lauten:

> *„Methods to facilitate the creation of String concatenation methods, that can be used to efficiently concatenate a known number of arguments of known types, possibly after type adaptation and partial evaluation of arguments. These methods are typically used as bootstrap methods for invokedynamic call sites, to support the string concatenation feature of the Java Programming Language."*

Im Gegensatz zu Lambda-Ausrücken, die als neu eingeführtes Sprachkonstrukt mit `invokedynamic` realisiert wurden, wird für die String-Konkatenation erst ab Java 9 `invokedynamic` verwendet. Der Leser kann sich daher mit `javac` und `javap` recht einfach davon überzeugen, dass der erzeugte Bytecode, zum Beispiel für Java 8 und 9, sich in diesem Punkt unterscheidet.

Records

Records wurden im JEP 359 als Preview in Java 14 eingeführt. Nach einem zweiten Preview mit JEP 384 in Java 15 wurden sie schließlich mit JEP 395 in Java 16 endgültig als Feature in die Sprache Java aufgenommen. Die Implementierung der Record-Methoden `equals()`, `hashCode()` und `toString()` erfolgte von Anbeginn an, also bereits mit Java 14, über `invokedynamic`. Die entsprechende Bootstrap-Methode ist `ObjectMethods#bootstrap()` im Package `java.lang.runtime` und dem folgenden *JavaDoc*:

Falls der Leser in das *JavaDoc* schaut, wird er feststellen, dass als erstmalige Version der Klasse 16 angegeben ist („since: 16"). Dies ist kein Widerspruch zum oben genannten JEP 359, der bereits mit Java 14 eingeführt wurde. Alle Klassen eines Previews werden im *JavaDoc* mit der Version des ersten Previews geführt. Wenn eine Klasse dann final in die Sprache Java aufgenommen wird, wird die Versionsnummer auf die finale Java-Version aktualisiert.

Pattern Matching for switch

Das Pattern-Matching für Switch-Ausrücke und -Anweisungen wurde mit JEP 406 als Preview in Java 17 eingeführt. Die JEPs 420, 427 und 433 wurden ebenfalls als jeweils nachfolgender Preview des Sprach-Features für die Java-Versionen 18, 19 und 20 definiert. Mit JEP 441 und Java 21 wurde diese Art des Pattern-Matching schließlich endgültig in die Sprache aufgenommen.

Die entsprechende Bootstrap-Methode ist `SwitchBootstrap#typeSwitch()` im Package `java.lang.runtime`. Der erste Satz des *JavaDoc* lautet:

> *„Bootstrap method for linking an invokedynamic call site that implements a switch on a target of a reference type."*

Die Klasse `SwitchBootstrap` enthält noch eine zweite Methode, nämlich `enumSwitch()`. Laut *JavaDoc* ist sie die Bootstrap-Methode, wenn der Switch-Ausdruck aus Werten eines Aufzählungstyps besteht. Wir konnten dies mit den Java-Versionen 17 bis 21 jedoch nicht nachstellen und können es somit hier nicht bestätigen.

String-Templates

String-Templates wurden mit JEP 430 [7] mit Java 21 als Preview-Feature eingeführt. Sie erlauben durch Template-Prozessoren die Auswertung von syntaktisch speziell geformten Ausdrücken innerhalb von Strings und Textblöcken. Auch hier wurde bei der Implementierung entschieden, diese auf `invokedynamic` zu basieren. Die drei Bootstrap-Methoden `newStringTemplate()`, `newLargeStringTemplate()` und `processStringTemplate()` sind in der Klasse `TemplateRuntime` im Package `java.lang.runtime` enthalten. Die ersten beiden Sätze des *JavaDoc* der Klasse lesen sich wie folgt:

> „Manages string template bootstrap methods. These methods may be used, for example, by Java compiler implementations to create StringTemplate instances."

Invoke Dynamic reloaded

Die Einführung von `invokedynamic` als neuer Bytecode-Befehl war, wie wir am zunächst ungeplanten, jetzt aber breiten Einsatzspektrum gesehen haben, ein Erfolg. Das Konzept, die Übersetzung eines Methodenaufrufs durch eine zusätzliche Indirektion zu flexibilisieren, scheint derart attraktiv, dass es auch für Java-Konstanten übernommen wurde. Der JEP 309, *Dynamic Class-File Constants* [8] hat genau dies zum Ziel und wurde mit Java 11 umgesetzt. Die Zusammenfassung lautet folgt:

„Extend the Java class-file format to support a new constant-pool form, CONSTANT_Dynamic. Loading a CONSTANT_Dynamic will delegate creation to a bootstrap method, just as linking an invokedynamic call site delegates linkage to a bootstrap method."

Hintergrund ist, dass der Bytecode für Konstanten nur einen kleinen Bereich und eingeschränkte Funktionalität vorsieht. Durch die Einführung der Klasse `LambdaMetafactory` verwenden deren Bootstrap-Methoden Parameter, die häufig Konstanten sind, die mit den herkömmlichen Konstrukten aufwendig zu realisieren sind. Mit dem JEP sollen diese Art von Konstanten flexibler erzeugt werden können.

Für die Umsetzung wurde die Klasse `ConstantBootstrap` im Package `java.lang.invoke` geschaffen, deren *JavaDoc*

„Bootstrap methods for dynamically-computed constants."

diese Flexibilität zum Ausdruck bring.

Zusammenfassung

Mit Java 7 wurde der neue Bytecode-Befehl `invokedynamic` eingeführt, um dynamische Sprachen, die auf der JVM ausgeführt werden sollen, einfacher implementieren zu können. Die zunächst nicht intendierte Verwendung zur Implementierung von Java-Sprach-Features erfolgte erstmalig für die mit Java 8 vorgestellten Lambda-Ausdrücke. In späteren Versionen wurden noch weitere neue Sprach-Features, aber etwa auch die String-Konkatenation, mithilfe von `invokedynamic` realisiert. Das Konzept hinter `invokedynamic` sieht vor, dass der aufzurufende Code nicht direkt in den Bytecode übersetzt, sondern über eine sogenannte Bootstrap-Methode die aufzurufende Call-Site zurückgeliefert wird, die dann aufgerufen wird. Diese zusätzliche Indirektion erlaubt in einer späteren Java-Version eine effizientere Ausführung, da der Bibliotheks-Code dieser verwendeten Java-Version eine bessere Umsetzung bereitstellen könnte , als die Java-Version, mit der der Code übersetzt wurde.

Es wurden bisher, Stand Java 21, die folgenden Klassen mit entsprechenden Bootstrap-Methoden implementiert:

- `java.lang.invoke.LambdaMetafactory`
- `java.lang.invoke.StringConcatFactory`
- `java.lang.runtime.ObjectMethods`
- `java.lang.runtime.SwitchBootstrap`
- `java.lang.runtime.TemplateRuntime`
- `java.lang.invoke.ConstantBootstrap`

Alle genannten Klassen und deren Methoden sind öffentlich, können also von Anwendungsentwicklern verwendet werden. Es gibt allerdings keine sinnvollen Verwendungsmöglichkeiten hierfür, da die Klassen einzig und allein als Werkzeugunterstützung für den Compiler entwickelt wurden.

Referenzen

[1] List of JVM languages. *https://en.wikipedia.org/wiki/List_of_JVM_languages*

[2] JSR 292: Supporting Dynamically Typed Languages on the Java Platform, *https://jcp.org/en/jsr/detail?id=292*

[3] Benjamin J. Evans, Martijn Verburg. The Well-Grounded Java Developer, Manning 2013

[4] Brian Goetz. Translation of Lambda Expressions, *https://bit.ly/3sLQxnw*

[5] Bernd Müller. Was jeder Java-Entwickler über Strings wissen sollte. Java aktuell 03/2023

[6] JEP 280: Indify String Concatenation, *https://openjdk.org/jeps/280*

[7] JEP 430: String Templates (Preview), *https://openjdk.org/jeps/430*

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.

Mitglieder des iJUG

01 BED-Con e.V.
02 Clojure User Group Düsseldorf
03 DOAG e.V.
04 EuregJUG Maas-Rhine
05 JUG Augsburg
06 JUG Berlin-Brandenburg
07 JUG Bremen
08 JUG Bielefeld
09 JUG Bonn
10 JUG Darmstadt
11 JUG Deutschland e.V.
12 JUG Dortmund
13 JUG Düsseldorf rheinjug
14 JUG Erlangen-Nürnberg
15 JUG Freiburg
16 JUG Goldstadt
17 JUG Görlitz
18 JUG Hannover
19 JUG Hessen
20 JUG HH
21 JUG Ingolstadt e.V.
22 JUG Kaiserslautern
23 JUG Karlsruhe
24 JUG Köln
25 Kotlin User Group Düsseldorf
26 JUG Mainz
27 JUG Mannheim
28 JUG München
29 JUG Münster
30 JUG Oberland
31 JUG Ostfalen
32 JUG Paderborn
33 JUG Saxony
34 JUG Stuttgart e.V.
35 JUG Switzerland
36 JSUG
37 Lightweight JUG München
38 SUG Deutschland e.V.
39 JUG Thüringen
40 JUG Saarland
41 JUG Duisburg
42 JUG Frankfurt

iJUG Verbund
www.ijug.eu

Impressum

Java aktuell wird vom Interessenverbund der Java User Groups e.V. (iJUG) (Tempelhofer Weg 64, 12347 Berlin, *www.ijug.eu*) herausgegeben. Es ist das User-Magazin rund um die Programmiersprache Java im Raum Deutschland, Österreich und Schweiz. Es ist unabhängig von Oracle und vertritt weder direkt noch indirekt deren wirtschaftliche Interessen. Vielmehr vertritt es die Interessen der Anwender an den Themen rund um die Java-Produkte, fördert den Wissensaustausch zwischen den Lesern und informiert über neue Produkte und Technologien.

Java aktuell wird verlegt von der DOAG Dienstleistungen GmbH, Tempelhofer Weg 64, 12347 Berlin, Deutschland, gesetzlich vertreten durch den Geschäftsführer Fried Saacke, deren Unternehmensgegenstand Vereinsmanagement, Veranstaltungsorganisation und Publishing ist.

DOAG e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. DOAG e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Björn Bröhl. DOAG e.V. informiert kompetent über alle Oracle-Themen, setzt sich für die Interessen der Mitglieder ein und führen einen konstruktiv-kritischen Dialog mit Oracle.

Redaktion:
Sitz: DOAG Dienstleistungen GmbH
ViSdP: Fried Saacke
Redaktionsleitung: Lisa Damerow
Kontakt: redaktion@ijug.eu

Redaktionsbeirat:
Andreas Badelt, Marcus Fihlon, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, Bennet Schulz

Titel, Gestaltung und Satz:
Alexander Kermas,
DOAG Dienstleistungen GmbH

Bildnachweis:
Titel: Bild © Designed by catalyststuff
https://freepik.com
S. 10 + 11: Bild © Designed by vectorpocket
https://freepik.com
S. 14: Bild © Sloth McSloth
https://stock.adobe.com
S. 16 + 17: Bild © ant
https://stock.adobe.com
S. 26 + 27: Bild © IBEX.Media
https://stock.adobe.com
S. 34 + 35: Bild © semisatch
https://stock.adobe.com
S. 42 + 43: Bild © Designed by upklyak
https://freepik.com
S. 50: Bild © Open Elements
https://open-elements.com
S. 56 + 57: Bild © Designed by stories
https://freepik.com
S. 64 + 65: Bild © Designed by macrovector
https://freepik.com
S. 70 + 71: Bild © PCH.Vector
https://stock.adobe.com
S. 76 + 77: Bild © Designed by freepik
https://freepik.com
S. 80 + 81: Bild © Designed by macrovector
https://freepik.com

Anzeigen:
DOAG Dienstleistungen GmbH
Kontakt: sponsoring@doag.org
Mediadaten und Preise:
www.doag.org/go/mediadaten

Druck:
WIRmachenDRUCK GmbH
www.wir-machen-druck.de

Inserentenverzeichnis