

Das IJUG Magazin

# Java aktuell

## Oracle und die Zukunft von Java

Erfahrungen, Ideen und Lösungen für Java-Entwickler





- 3 Editorial  
*Wolfgang Taschner*
- 5 Die Oracle + Sun Java-Strategie  
*Oracle Corporation*
- 6 Interview mit Richard Seibt:  
„Oracle könnte durch richtige Strategien und entsprechende Investments die Innovationsgeschwindigkeit von Java noch erhöhen ...“
- 9 Interview mit Günther Stürner und Bernd Rintelmann:  
„Sun ist als Marke wertvoll, sie stand und steht für Innovation, Geschwindigkeit und IT-Kultur erster Güte ...“
- 13 Oracle und die Zukunft von Java  
*Markus Eisele*
- 16 Die Geschichte der Bohne  
*Oliver Szymanski*
- 19 JSP – das etwas andere Web Framework  
*David Tanzer*
- 23 Enterprise Java im Griff des Spring Frameworks  
*Robert Szilinski*
- 27 Ausblick auf Java 7  
*Oliver Szymanski*
- 30 Aus Alt mach Neu  
*Markus Eisele*
- 33 Entwicklungsumgebung mit NetBeans, GlassFish v3 und PostgreSQL  
*Gunther Petzsch*
- 37 Source Talk Tage in Göttingen  
*Stefan Koospal*
- 39 Clojure  
*Stefan Kamphausen*
- 41 Pleiten, Pech und PatternTesting – ein Drama in mehreren Akten  
*Oliver Böhm*
- 46 Der Werkzeugkasten des Admins für den Java-Server  
*Tobias Frech*
- 50 JUnit – Tests lesbar gestalten  
*Dirk Dittert*
- 55 Java everywhere  
*Till Hahndorf*
- 54 Inserentenverzeichnis
- 56 Impressum
- 62 Apex und PL/SQL meet Java: unbegrenzte Möglichkeiten mit der Datenbank-JVM  
*Carsten Czarski*
- 66 Termine der Java-Community



Oracle-Mitarbeiter stellen sich den Fragen des iJUG  
Seite 9



Netbeans-Training im Rahmen der Source Talk Tages  
Seite 60



Die globale Welt von Java am Beispiel der JUG Belgrad  
Seite 55

**Dies ist ein Sonderdruck aus der Java aktuell. Er enthält einen ausgewählten Artikel aus der Ausgabe 04/2010. Das Veröffentlichen des PDFs bzw. die Verteilung eines Ausdrucks davon ist lizenzfrei erlaubt. Weitere Informationen unter [www.ijug.eu](http://www.ijug.eu)**



# Der Werkzeugkasten des Admins für den Java-Server

Tobias Frech, Frech.IT

Entwickelte Java-Anwendungen müssen auch betrieben werden. Jede Java Virtual Machine (JVM) stellt dafür eine Reihe von Werkzeugen zur Verfügung. Der Artikel stellt die Administrationswerkzeuge der Sun JVM vor, die für den Einsatz eines Java-Servers notwendig sind.

Wenn die Java-Anwendung auf dem Server immer wieder kleine „Denkpausen“ einlegt oder den Betrieb sporadisch mit einem OutOfMemory-Error quittiert, fragt sich der Admin: „Woran liegt das?“ Mithilfe der Standardwerkzeuge des Betriebssystems wird er versuchen, den Java-Server-Prozess zu untersuchen und die Ursache des Problems zu ermitteln. Die Sun JVM hält zusätzliche Werkzeuge zur Diagnose bereit, mit deren Hilfe sich in den laufenden Java-Prozess hineinschauen lässt. Diese Werkzeuge werden nachfolgend anhand von typischen Administrationsproblemen vorgestellt.

Die meisten Informationen bieten sicherlich die beiden grafischen Werkzeuge JConsole und VisualVM. Diese sind jedoch nicht auf jedem Server einsetzbar, insbesondere, wenn er „remote“ administriert wird. Sie setzen zudem eine gewisse Kenntnis der Materie voraus. Wir wenden uns daher in diesem Artikel den Grundlagen und den einfachen Kommandozeilen-basierten Werkzeugen zu. Dazu muss ein Sun Java Development Kit (JDK) installiert sein; das Java Runtime Environment (JRE) bringt diese Werkzeuge nicht mit. Im Verzeichnis, in dem sich die JVM „java“ befindet, sind die Kommandozeilen-Werkzeuge `jps`, `jstat/jstatd`, `jinfo`, `jmap` und

```
[tobi@fedora ~]$ jps
13257 Jps
13199 Main

[tobi@fedora ~]$ jps -l -m
13199 org.jboss.Main -c minimal
13244 sun.tools.jps.Jps -l -m

[tobi@fedora ~]$ jps -l -v
13199 org.jboss.Main -Dprogram.name=run.sh -Xms128m -Xmx150m -Dsun.rmi.dgc.client.gcInterval=3600000 -Dsun.rmi.dgc.server.gcInterval=3600000 -Xloggc:gc.log -Djava.net.preferIPv4Stack=true -Djava.endorsed.dirs=/home/tobi/JBoss/jboss-4.2.3.GA/lib/endorsed
13232 sun.tools.jps.Jps -Dapplication.home=/usr/java/jdk1.6.0_16 -Xms8m
```

Abbildung 2: `jps` listet alle laufenden Java-Prozesse auf, hier ein JBoss-Server

`jstack` abgelegt (siehe <http://java.sun.com/javase/6/docs/technotes/tools/index.html#monitor>). Zum JDK sind sie seit der Version 5 hinzugekommen. Insbesondere unter Windows sind diese Werkzeuge aber nur eingeschränkt oder teilweise vorhanden. Abbildung 1 zeigt die genaue Übersicht. Die folgenden Beispiele sollen dazu animieren, selbst auszuprobieren und für die eigene Situation nützliche Werkzeuge zu entdecken.

## jps: Anwendungen finden

Jedes der Werkzeuge erwartet als einen Parameter die sogenannte „`lvmid`“, die

üblicherweise die Prozess-ID des Java-Prozesses darstellt. Diese lässt sich über `jps` ermitteln (Merkhilfe für Unix-Admins: „`java ps`“). Es werden nur die Java-Prozesse aufgelistet, die für den aktuellen Benutzer gemäß den Systemrechten erreichbar sind. In der Liste wird auch `jps` mit ausgegeben, da die Kommandozeilen-Werkzeuge selbst in Java realisiert sind. Mit der Option „`-v`“ lassen sich alle Parameter ausgeben, die der JVM übergeben wurden (etwa für die Speicher und GC-Einstellungen). Mit der Option „`-l`“ wird der vollständige Name der Main-Klasse (zum Beispiel `org.jboss.Main` für einen JBoss-Server) ausgegeben und die Option „`-m`“ zeigt alle Parameter, die an die Main-Klasse als Parameter weitergereicht wurden (beispielsweise die Server-Configuration bei einem JBoss-Server). Abbildung 2 zeigt eine solche Ausgabe.

## jinfo: JVM-Konfiguration auslesen

Ähnlich wie die Environment-Variable eines Unix-Prozesses hat auch jede JVM ihr

Werkzeug	JDK 5	JDK 6
<code>jps</code>	ok	ok
<code>jinfo</code>	kein Windows/Linux Itanium	Windows eingeschränkt
<code>jstat</code>	ok	ok
<code>jstack</code>	kein Windows/Linux Itanium	Windows eingeschränkt
<code>jmap</code>	kein Windows/Linux Itanium	Windows eingeschränkt

Abbildung 1: Die JDK-Werkzeuge unterscheiden sich je nach Betriebssystem in der Verfügbarkeit und dem Funktionsumfang



eigenes internes „Environment“: die System Properties. Um Klarheit über die eigentliche Konfiguration einer JVM zu bekommen, sind nicht nur die Parameter beim JVM-Start erheblich (etwa „-Xmx5g“ für den maximalen Heap-Speicher). Auch die Inhalte eben dieser System Properties sind relevant, die entweder beim JVM-Start mit der „-D“-Option gesetzt werden können (beispielsweise „-Djava.awt.headless=true“ für Server mit Grafik-Verarbeitung), programmatisch durch die JVM oder die durch Java-Anwendung selbst. Mit `jinfo` lassen sich alle Schlüssel-Wert-Paare der System Properties anzeigen. Dies kann dann nützlich werden, wenn beispielsweise unklar ist, mit welcher Timezone oder Locale die jeweilige JVM arbeitet. Abbildung 3 zeigt ein Beispiel dazu.

Mit der Option „-flag“ kann sogar ein Kommandozeilen-Parameter nachträglich gesetzt werden, um zum Beispiel automatische Heap Dumps bei Speicherproblemen zu aktivieren:

```
jinfo -flag +HeapDumpOnOutOfMemoryError <lvmid>
jinfo -flag HeapDumpPath=/tmp/<lvmid>
```

Nicht jede Einstellung der JVM lässt sich jedoch nachträglich, das heißt nach dem Start, verändern.

### jstat: Garbage Collection und andere Laufzeit-Informationen

Eines der wertvollsten Werkzeuge im Werkzeugkasten des Java-Administrators ist `jstat`. Mit diesem Werkzeug lassen sich aktuelle Informationen über die Anzahl der geladenen Klassen, die Aktivitäten des Just-in-Time-Compilers, die Garbage-Collection-Aktivitäten und die Speicherverwaltung ausgeben. In der JDK-Dokumentation sind alle zwölf Optionen zu finden, welche die jeweils auszugebenden Informationen steuern. Ebenfalls findet man dort die Bedeutung der verschiedenen Spaltenüberschriften in der Ausgabe. Hier soll nun näher auf die Ausgabe der Option „-gc“ eingegangen werden. Abbildung 4 zeigt die Ausgabe eines Aufrufs mit der `lvmid` zusammen ohne weitere Optionen.

Zur besseren Übersichtlichkeit wurden die Spaltenüberschriften zusammen mit dem zugeordneten Inhalt umbrochen. Die

```
[tobi@fedora ~]$ jstat -gc 14242
SOC      S1C      SOU      S1U      EC      EU
9856,0   7296,0   0,0      7288,4   90496,0 32828,6

OC      OU      PC      PU
28480,0 20606,8 32896,0 32826,9

YGC      YGCT     FGC      FGCT     GCT
13       0,461    3        0,381    0,842
```

Abbildung 4: Ausgabe eines Aufrufs mit der `lvmid`

```
[tobi@fedora ~]$ jinfo 14042 | grep user.
Attaching to process ID 14042, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 1.5.0_17-b04
user.name = tobi
user.language = de
user.timezone = Europe/Berlin
user.country = DE
user.home = /home/tobi
user.dir = /home/tobi/JBoss/jboss-4.2.3.GA/bin
```

Abbildung 3: `jinfo` gibt Auskunft über die Inhalte der System Properties der JVM

Überschriften der Spalten haben folgende Bedeutungen:

- S0C/S1C: Survivor Space 0 bzw. 1 Capacity (also aktuelle Größe)
- S0U/S1U: Survivor Space 0 bzw. 1 Utilization (also Füllstand des Bereichs)
- EC und EU: Eden Capacity und Utilization
- OC und OU: Old Generation Capacity und Utilization
- PC und PU: Permanent Generation Capacity und Utilization
- YGC und YGCT: Anzahl und kumulative Zeit der Minor (Young) Garbage Collection
- FGC und FGCT: Anzahl und kumulative Zeit der Full Garbage Collection
- GCT: kumulative Zeit der Garbage Collection gesamt (Minor + Full GC)

Eine genaue Beschreibung der Funktionsweise der Garbage Collection (GC) findet sich in einem White Paper von Sun unter [http://java.sun.com/j2se/reference/whitepapers/memorymanagement\\_whitepaper.pdf](http://java.sun.com/j2se/reference/whitepapers/memorymanagement_whitepaper.pdf). Es lassen sich jedoch auch leicht einige grundlegende Zusammenhänge an den obigen Werten beobachten. Die beiden Survivor Spaces zusammen mit

dem Eden-Bereich bilden die sogenannte „Young Generation“. Neue Objekte werden im Eden-Bereich abgelegt. Eine Minor GC (Zähler: YGC) räumt diese Bereiche auf und sollte zu einem leeren Eden-Bereich führen. Gleichzeitig können langlebigere Objekte in die Old Generation verschoben werden (OC und OU erhöhen sich). Ist die Old Generation voll (OU erreicht 100), so wird spätestens dann eine Full GC (Zähler: FGC) ausgelöst. Dass sowohl die Young als auch die Old Generation immer wieder bis zu 100 Prozent gefüllt werden, ist in einer JVM vollkommen normal. Wichtig ist, dass nach der jeweilig zugeordneten GC wieder Speicher freigegeben werden konnte, das heißt, der jeweilige Utilization-Wert ist dann Null oder sinkt merklich. Es lassen sich nun die drei folgenden Effekte beobachten:

1. Wenn die Anwendung immer wieder eine merkliche Pause macht, so könnte dies an einer „Full GC“ liegen. Während einer GC (Minor und Full) wird in der Grundeinstellung die Anwendung pausiert, bis die GC abgeschlossen ist. Minor GCs sind im Allgemeinen sehr schnell durchgeführt. Full GCs können jedoch je nach Speichergröße von



mehreren Gigabyte und abhängig von der Speichergeschwindigkeit und Prozessoranzahl durchaus auch mehrere Sekunden bis zu Minuten in Anspruch nehmen. Solche Pausen wären in deutlichen Sprüngen in der kumulativen Zeit der Full GC sichtbar (Zähler: FGCT in Sekunden). Abhilfe kann hier schaffen, eine GC-Strategie zu aktivieren, die während der Full GC mehrere Prozessoren nutzt oder die parallel zur Anwendungsausführung angewendet wird. Beide Wege sind nicht unproblematisch und müssen sorgfältig geplant, konfiguriert und getestet werden.

- Die Anwendung wird mit der Zeit merklich langsamer, bis dann die aktuelle Verarbeitung mit einem OutOfMemory-Error abgebrochen wird. In dieser Situation wird der Speicher der JVM zu knapp, nach einer Full GC kann kaum noch Speicher freigegeben werden und die nächste Full GC folgt in relativ kurzem Zeitabstand. Bei normalem Betrieb sollte die Minor GC deutlich häufiger auftreten als die Full GC, das heißt, YGC hat einen deutlich höheren Wert als FGC. Tritt die oben beschriebene Situation ein, so erhöht sich fast nur noch der FGC-Zähler und letztendlich nimmt die kumulative Zeit der GC (Zähler: GCT) genau so schnell zu, wie die reale Zeit verstreicht. Die JVM beschäftigt sich dann fast ausschließlich mit der GC. Nach einiger Zeit führt dies zum OutOfMemory-Error. Hier lässt sich ein gutes Monitoring ansetzen, das dieses beschriebene Muster erkennt und frühzeitig warnt, bevor der Speicher endgültig zu voll ist.
- Sowohl die Young als auch die Old Generation werden durch die GC immer wieder frei geräumt. Die Anwendung steigt

jedoch trotzdem mit einem OutOfMemory-Error aus. Hier lohnt ein Blick auf die Utilization der Permanent Generation (Zähler: PU). Steigt dieser Wert selbst nach einer Full GC immer weiter an, so ist die Permanent Generation eventuell zu klein gewählt. Tückischerweise wird dieser Speicherbereich nicht mit dem Parameter „-Xmx1500m“ gesteuert, sondern mit „-XX:MaxPermSize=512m“.

Damit man zur Beobachtung der JVM das jstat-Kommando nicht immer wieder absetzen muss, gibt es die Möglichkeit, die Werte zyklisch wiederholt ausgeben zu lassen. Dazu gibt man als zusätzlichen Parameter die Zeit in Millisekunden oder in Sekunden mit der Endung „s“ an (siehe Abbildung 5).

#### jstack:

#### Was macht die Anwendung eigentlich?

Wenn unklar ist, mit welchen Aufgaben sich die Anwendung momentan beschäftigt, so kann auf verschiedenen Wegen ein sogenannter „Thread-Dump“ für die gesamte JVM erzeugt werden. Dabei wird für jeden einzelnen Thread der momentane Stand des Call-Stacks ausgegeben. Abbildung 6 zeigt einen kurzen Beispielausschnitt. Jede einzelne Zeile in diesem Stack beginnt mit dem Objektname und der konkreten Methode innerhalb des Objekts, die gerade abgearbeitet wird. In Klammern wird dann noch der Name der Quellcodedatei mit Angabe der Zeilennummer in dieser Datei ausgegeben. Der Administrator kann aus den reinen Objekt- und Methodenname natürlich nicht schließen, was genau an dieser Codestelle passiert. Jedoch kann der Anfang des Objektname – das sogenannte „Package“ – sehr wohl deutliche Hinweise darauf geben, in welcher Schicht der Anwendung der Programmteil

angesiedelt ist. So ist zum Beispiel leicht erkennbar, wenn der entsprechende Code aus dem JDBC-Datenbanktreiber stammt (für Oracle beginnt der Package-Name beispielsweise mit oracle.jdbc). Beim Lesen des Call-Stacks ist darauf zu achten, dass die unterste Zeile der eigentliche Start ist und jede Zeile darüber einen Methodenaufruf innerhalb der Bearbeitung darstellt, das heißt, dass die oberste Zeile eines Call-Stacks die aktuelle Position der Programmabarbeitung für diesen Thread darstellt. Mit dem Thread-Dump für alle Threads lässt sich schnell ein groben Überblick darüber verschaffen, mit welchen Funktionen die verschiedenen Threads im Server beschäftigt sind.

Um nun einen solchen Thread-Dump zu erhalten, kommt das Werkzeug jstack zum Einsatz. Nach Aufruf mit der entsprechenden lvmid wird der Thread-Dump erzeugt und ausgegeben. Der Server läuft nach dieser Ausgabe üblicherweise normal weiter.

Wird diese Ausgabe wiederholt angefordert, so kann man sich die statistische Eigenschaft zunutze machen, dass die Codestellen, die am meisten Zeit „verbrauchen“, am häufigsten in einem Thread-Dump auftauchen sollten. jstack ist sozusagen der Profiler des Admins, der nicht zu stark in eine Produktiv-Umgebung eingreifen will oder darf und der sich den Gang zur Beschaffungsabteilung ersparen will. Das Verfahren, sich wiederholt Thread-Dumps von der JVM zu holen, wird übrigens vom Sampler-Plugin des VisualVM-Werkzeugs eingesetzt, um eine gewohnte Ausgabe eines Profilers zu erzeugen. Dies funktioniert sogar mit Produktivsystemen unter Last erstaunlich gut.

#### jmap: Speicherverschwendung aufspüren

Gerade wenn der Speicher knapp geworden ist beziehungsweise schon nicht mehr ausgereicht hat, stellt sich die Frage, wodurch dieser verbraucht wird. Eine Analyse auf einem Entwicklungs- oder Testsystem kann nur in manchen Fällen den Verursacher ermitteln, der auch im produktiven Betrieb den Speicherverbrauch nach oben treibt.

Bei vielen Anwendungen im laufenden Betrieb ist in einer solchen Situation für eine Diagnose des Speichers eine Un-

```
[tobi@fedora ~]$ jstat -gcutil 14343 2s
```

S0	S1	E	O	P	YGC	YGCT	FGC	FGCT	GCT
0,00	39,14	38,86	63,89	72,26	33	0,628	7	1,161	1,789
60,22	0,00	48,74	63,89	72,27	34	0,648	7	1,161	1,809
46,10	0,00	39,14	65,21	72,29	36	0,691	7	1,161	1,852
50,88	0,00	15,55	65,68	72,30	38	0,717	7	1,161	1,878
0,00	83,74	83,53	65,84	72,32	39	0,727	7	1,161	1,888
0,00	98,73	85,56	66,17	72,32	41	0,750	7	1,161	1,911

Abbildung 5: Zyklische Ausgabe



```
„pool-1-thread-2“ prio=10 tid=0x2b670400 nid=0x264c runnable [0x2b3ad000]
  java.lang.Thread.State: RUNNABLE
    at java.lang.String.intern(Native Method)
    at org.apache.lucene.document.Field.<init>(Field.java:285)
    at org.apache.lucene.index.FieldsReader.addField(FieldsReader.java:357)
    at org.apache.lucene.index.FieldsReader.doc(FieldsReader.java:197)
    at org.apache.lucene.index.SegmentReader.document(SegmentReader.java:733)
  ...
    at org.apache.lucene.search.Searcher.search(Searcher.java:105)
    at info.frech.fema.SearchService.searchInternal(SearchService.java:77)
    at info.frech.fema.SearchService.search(SearchService.java:42)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  ...
    at com.sun.net.httpserver.Filter$Chain.doFilter(Filter.java:65)
    at sun.net.httpserver.ServerImpl$Exchange.run(ServerImpl.java:527)
    at java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:886)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:908)
    at java.lang.Thread.run(Thread.java:619)
```

Abbildung 6: Ausschnitte aus einem Call-Stack für einen Thread

terbrechung, die nur wenige Sekunden dauert, noch vertretbar. Dann kann das Werkzeug jmap mit der Option „-histo“ eingesetzt werden, um ein Histogramm zu erzeugen, welches pro geladener Klasse die Anzahl der Objektinstanzen anzeigt (siehe Abbildung 7).

Neben der Anzahl der Instanzen wird auch noch die Speichergröße ausgegeben, die durch die Instanzen selbst belegt wird. Hierbei ist es nicht ungewöhnlich, dass Arrays (Prefix []) von Bytes ([B]), Integern ([I]) oder Chars ([C]) den meisten Speicher verbrauchen. Allerdings sind eigentlich die Klassen von Interesse, welche diese Arrays im Speicher referenzieren und damit im Speicher halten. Ein solches Histogramm kann also nur mit entsprechender Kenntnis, wie eine Software intern auf Klassenebene funktioniert, richtig interpretiert und dadurch Auffälligkeiten festgestellt werden. Hilfreich kann es hierbei auch sein, wenn über die Zeit immer wieder Histogramme erstellt werden und sich so Trends erkennen lassen. Wenn die Unterbrechungen noch ein wenig länger sein dürfen, so ist die Option „-histo:live“ zu bevorzugen, die vor der Histogramm-Erstellung noch eine Full Garbage Collection ausführt und damit nur Klassen anzeigt, die in diesem Moment noch erreichbar und damit noch aktiv sind.

Soll die Analyse des Speicherinhalts ausführlich ausfallen, so kann mit jmap auch ein Speicherabbild (Dump) erzeugt werden:

```
[tobi@fedora ~]$ jmap -histo 3649
```

num	#instances	#bytes	class name
1:	584317	612516808	[B
2:	3618	23825536	[I
3:	400239	17117128	[C
4:	29569	14920848	[Ljava.lang.Object;
5:	397347	9536328	java.lang.String
6:	168406	4041744	java.util.LinkedList\$Entry
7:	111696	2680704	info.frech.fema.Feature
8:	24033	2444688	<constMethodKlass>
9:	24033	1926520	<methodKlass>
10:	34899	1596104	<symbolKlass>
11:	2269	1138096	<constantPoolKlass>
12:	2269	897264	<instanceKlassKlass>
13:	27924	893568	info.frech.fema.History\$Entry
14:	26219	839008	org.apache.lucene.index.TermInfo
15:	2060	731328	<constantPoolCacheKlass>
16:	28364	680736	java.util.ArrayList
17:	28313	679512	java.util.LinkedList
18:	27924	670176	java.util.Date
...	...	...	...

Abbildung 7: Ergebnis der Option „-histo“

```
jmap
-dump:live,format=b,file=heap.
hprof 5197
```

Je nach Größe des momentan verwendeten Speichers dauert diese Aktion im Bereich von Sekunden bis zu Minuten. Die Anwendung wird für diese Dauer angehalten. In einer produktiven Umgebung ist also die Erzeugung eines Speicherabbaus aufgrund der Unterbrechung nur

eingeschränkt möglich. Wurde jedoch ein solches Abbild erzeugt, so kann mit Kenntnis der Funktionsweise der Software und einem entsprechenden Werkzeug der Speicherverbrauch analysiert werden. Empfehlenswert ist hierfür der Eclipse Memory Analyzer (MAT), der aber nicht mehr Teil der JDK-Tools ist.

**Kontakt:**  
Tobias Frech  
tobias@frech.info



## Bestellen eines kostenlosen Exemplares der Zeitschrift Java aktuell

Anschrift:

\_\_\_\_\_  
Name, Vorname

\_\_\_\_\_  
Firma

\_\_\_\_\_  
Abteilung

\_\_\_\_\_  
Straße, Hausnummer

\_\_\_\_\_  
PLZ, Ort

\_\_\_\_\_  
ggf. Rechnungsanschrift

\_\_\_\_\_  
E-Mail

\_\_\_\_\_  
Telefonnummer

\_\_\_\_\_  
Die allgemeinen Geschäftsbedingungen\* erkenne ich an, Datum, Unterschrift

## Jetzt Abonnement sichern:

- Abonnement Newsletter: Java aktuell – der iJUG-Newsletter, kostenfrei
- Java aktuell – das iJUG-Magazin Abo: vier Ausgaben zu 18 Euro im Jahr

Für Oracle-Anwender und Interessierte gibt es das Java aktuell Abonnement auch mit zusätzlich sechs Ausgaben im Jahr der Fachzeitschrift DOAG News und zwei Ausgaben im Jahr Business News. Weitere Informationen unter [www.doag.org/shop/](http://www.doag.org/shop/)

Senden Sie das ausgefüllte Formular an:

Interessenverbund der Java User Groups e.V.  
Tempelhofer Weg 64  
12347 Berlin

oder faxen Sie es an:

0700 11 36 24 39

oder bestellen Sie online:

[go.ijug.eu/go/abo](http://go.ijug.eu/go/abo)

\*Allgemeine Geschäftsbedingungen:

Zum Preis von 18 Euro (inkl. MwSt.) pro Kalenderjahr erhalten Sie vier Ausgaben der Zeitschrift "Java aktuell - das iJUG-Magazin" direkt nach Erscheinen per Post zugeschickt. Die Abonnementgebühr wird jeweils im Januar für ein Jahr fällig. Sie erhalten eine entsprechende Rechnung. Abonnementverträge, die während eines Jahres beginnen, werden mit 4,90 Euro (inkl. MwSt.) je volles Quartal berechnet. Das Abonnement verlängert sich automatisch um ein weiteres Jahr, wenn es nicht bis zum 31. Oktober eines Jahres schriftlich gekündigt wird. Die Wiederrufsfrist beträgt 14 Tage ab Vertragserklärung in Textform ohne Angabe von Gründen.

## Impressum

Herausgeber:  
Interessenverbund der Java User  
Groups e.V. (iJUG)  
Tempelhofer Weg 64, 12347 Berlin  
Tel.: 0700 11 36 24 38  
[www.ijug.eu](http://www.ijug.eu)

Verlag:  
DOAG Dienstleistungen GmbH  
Fried Saacke, Geschäftsführer  
[info@doag-dienstleistungen.de](mailto:info@doag-dienstleistungen.de)

Chefredakteur (VisdP):  
Wolfgang Taschner,  
[redaktion@ijug.eu](mailto:redaktion@ijug.eu)

Chefin von Dienst (CvD):  
Carmen Al-Youssef,  
[office@ijug.eu](mailto:office@ijug.eu)

Titel, Gestaltung und Satz:  
Claudia Wagner,  
DOAG Dienstleistungen GmbH

Anzeigen:  
CrossMarkeTeam, Ralf Rutkat,  
Doris Budwill  
[redaktion@ijug.eu](mailto:redaktion@ijug.eu)

Mediadaten und Preise:  
[http://www.ijug.eu/images/  
vorlagen/2011-ijug-mediadaten\\_  
java\\_aktuell.pdf](http://www.ijug.eu/images/vorlagen/2011-ijug-mediadaten_java_aktuell.pdf)

Druck:  
adame Advertising and Media  
GmbH Berlin  
[www.adame.de](http://www.adame.de)

**Java aktuell – das Abo**  
**4 Ausgaben für 18 Euro**