Serialisierung

Aus der Java-Trickkiste: Java-Serialisierung – wann passt sie, wann nicht?

7. März 2016 Arno Haase

Serialisierung ist ein Mechanismus, bei dem Objekte in eine Folge von Bytes verwandelt und umgekehrt daraus wieder Objekte erzeugt werden. Man braucht solche Mechanismen beispielsweise für das Aufrufen über ein Netzwerk oder um Objekte in einer Datenbank zu speichern. Java bringt dafür von Haus aus einen Mechanismus mit: die Serialisierung im engeren Sinne. Die ist trotz ihrer Schwächen so weit verbreitet, dass wir sie heute näher betrachten.

Grundlagen zur Serialisierung

Die größte Stärke der Java-Serialisierung ist, dass sie sehr einfach zu verwenden ist. Wenn eine beliebige Klasse das Interface *java.io.Serializable* implementiert, kann Java deren Instanzen serialisieren (Listing 1). Dabei muss man kein besonderes Programmiermodell beachten – es funktioniert z. B. auch mit final-Feldern und ohne

Default-Konstruktor, weil die Deserialisierung Objekte ohne Konstruktoraufruf erzeugt und anschließend die Felder über Reflection setzt. Und *Serializable* ist ein reines Marker-Interface ohne Methoden, die man implementieren müsste – viel einfacher geht es nicht.

Listing 1

```
1  public class Person implements Serializable {
2    private final String name;
3
4    public Person (String name) {
5      this.name = name;
6    }
7
8    public String getName () {
9      return name;
10   }
11 }
```

Zum eigentlichen Serialisieren gibt es die Klasse *java.io.ObjectOutputStream* (Listing 2). Der Code erzeugt eine Person-Instanz und schreibt sie in die Datei *dummy.ser*. Dazu erzeugt er mit der *try-with-resource-Syntax (http://bit.ly/1xCmiL1)*, die Java seit Version 1.7 unterstützt [1], einen *FileOutputStream* und einen *ObjectOutputStream* als Wrapper um diesen herum.

Listing 2

```
1  Person p = new Person("Arno");
2  try (FileOutputStream fos = new FileOutputStream ("dummy.ser"
3        ObjectOutputStream oos = new ObjectOutputStream (fos)) {
4    oos.writeObject (p);
5  }
```

Dann ruft er die Methode *writeObject()* auf dem *ObjectOutputStream* mit der *Person*-Instanz als Argument auf. Die wandelt das Objekt intern in eine Bytefolge um und schreibt diese an den *FileOutputStream* weiter – fertig ist die Serialisierung, das Objekt steht in der Datei. Und weil *ObjectOutputStream* als Wrapper um einen beliebigen anderen *OutputStream* funktioniert, kann man Objekte genauso leicht z. B. über eine Netzwerkverbindung (mit *Socket.getOutputStream()*) oder in ein Bytearray (mit *ByteArrayOutputStream*) übertragen.

Das Einlesen von Objekten funktioniert analog zum Schreiben, indem man einen *ObjectInputStream* um einen beliebigen anderen *InputStream* wickelt (Listing 3). Die

Methode *readObject()* holt dabei so lange Bytes aus der Datei, bis sie daraus ein fertiges Objekt erzeugen kann, und liefert das zurück – der Code muss natürlich noch auf *Person* casten.

Listing 3

```
try (FileInputStream fis = new FileInputStream ("dummy.ser");
    ObjectInputStream ois = new ObjectInputStream (fis)) {
  final Person p2 = (Person) ois.readObject ();
  assert (p2.getName ().equals ("Arno"));
}
```

Durch Hintereinanderschalten von Serialisierung und Deserialisierung kann man recht einfach eine generische Methode zum tiefen Kopieren von Objekten schreiben (Listing 4). Sie ist zwar nicht besonders effizient, erfordert dafür aber praktisch keine Kooperation der zu klonenden Objekte. Sie kann nützlich sein, um in Tests verteilte Aufrufe zu mocken, wo Änderungen an Objekten im Server nicht unmittelbar auf Objekte im Client wirken.

Listing 4

```
static <T> T genericClone (T o) throws IOException, ClassNot
  final ByteArrayOutputStream baos = new ByteArrayOutputStre
  try (ObjectOutputStream oos = new ObjectOutputStream (baos
    oos.writeObject (o);
  }

  ByteArrayInputStream bais = new ByteArrayInputStream (baos
  try (ObjectInputStream ois = new ObjectInputStream (bais))
    return (T) ois.readObject ();
  }
}
```

Identität von Objekten bleibt erhalten

Java berücksichtigt beim Serialisieren ganze Objektnetze. Wenn es eine Referenz auf ein anderes Objekt gibt, wird das mit serialisiert. Die einzige Voraussetzung ist, dass alle beteiligten Klassen *Serializable* implementieren. Dabei bleibt die Objektidentität innerhalb eines *ObjectOutputStreams* erhalten: Wenn es mehrere Referenzen auf dasselbe Objekt gibt, zeigen auch nach dem Deserialisieren die Referenzen auf dasselbe Objekt.

Listing 5 zeigt das am Beispiel einer Liste mit drei Personen. Die ersten beiden Einträge zeigen auf dasselbe Objekt, während der dritte auf ein anderes Objekt mit dem gleichen Inhalt zeigt. Und das ist nach einem Serialisierungs- und Deserialisierungsdurchlauf immer noch so.

```
Person p = new Person("Arno");
List<Person> persons = Arrays.asList (p, p, new Person("Arno"

List<Person> cloned = genericClone (persons);

assert (cloned.get(0) == cloned.get(1));
assert (cloned.get(0) != cloned.get(2));
```

Schwieriger wird es, wenn die Identität eines Objekts global erhalten bleiben soll. Listing 6a zeigt eine Variante der *Person*-Klasse, die eine *public static final*-Variable mit einer *THE_BOSS*-Instanz hat und sicherstellt, dass nur diese eine Instanz den Namen „The Boss“ haben kann. Das Beispiel ist natürlich konstruiert, aber es gibt in der Praxis tatsächlich Fälle wie diesen – sie haben aber typischerweise mehr Kontext und taugen deshalb nicht so gut als Beispiel.

```
public class Person implements Serializable {
    public static final Person THE_BOSS = new Person ("The Bos
    private final String name;

    private Person (String name) {
      this.name = name;
    }

    public static Person create (String name) {
      if (name.equals (THE_BOSS.getName ())) return THE_BOSS;
      return new Person (name);
    }

    public String getName () {
      return name;
    }
  }
```

Diese Klasse hat allerdings ein Problem: Wenn man *THE_BOSS* serialisiert und wieder deserialisiert, dann erhält man eine Kopie. Und damit ist die Invariante gebrochen, dass es nur einen Boss geben kann.

Listing 6b

```
1  public class Person implements Serializable {
2    ...
3    public Object readResolve() {
4      if (name.equals (THE_BOSS.getName ())) return THE_BOSS;
5      return this;
6    }
7  }
```

Für solche Situationen gibt es die Möglichkeit, beim Deserialisieren quasi einzugreifen. Wenn eine Klasse eine Methode *readResolve()* implementiert, die *public* ist, den Rückgabetyp *Object* hat und keine Parameter erwartet, ruft Java diese Methode bei jedem Deserialisieren auf (Listing 6b). Dabei wird das Objekt zunächst komplett normal deserialisiert und anschließend auf diesem eigentlich fertigen Objekt *readResolve()* aufgerufen, deren Rückgabewert dann als deserialisiertes Objekt weiter verwendet wird.

Eine Implementierung kann im einfachsten Fall *this* zurückgeben, dann verhält sich die Deserialisierung unverändert. Sie kann vorher das zurückgegebene Objekt auch speichern, es irgendwo registrieren, initialisieren etc. Oder sie kann eine komplett andere Instanz zurückgeben, z. B. das *THE_BOSS*-Singleton.

Dieses Beispiel ist mit Abstand der häufigste Anwendungsfall für *readResolve()*-Methoden. Java-Enums verwenden sie z. B. auf diese Weise. Es gibt analog zu *readResolve()* auch die Möglichkeit, durch Implementieren einer Methode *writeReplace()* das zu serialisierende Objekt auszutauschen. Mir ist allerdings noch kein Anwendungsfall begegnet.

Customizing, um keine Bandbreite zu verschwenden

Per Default serialisiert Java alle nicht statischen Felder eines Objekts. Das funktioniert, ist aber manchmal überflüssig und verschwendet Platz bzw. Bandbreite. Nehmen wir beispielsweise eine Variante der Klasse *Person* an, die sowohl den Vor- als auch den Nachnamen kennt und außerdem als Optimierung redundant den vollen Namen im Konstruktor erzeugt und in einem Feld speichert (Listing 7). Dieses Feld braucht man beim Serialisieren nicht mit zu übertragen, weil man es ja beim Deserialisieren aus Vor- und Nachnamen wieder ermitteln kann.

Listing 7

```
1  public class Person implements Serializable {
2    private final String firstName;
3    private final String lastName;
4    private transient final String name;
5
6    public Person (String firstName, String lastName) {
7      this.firstName = firstName;
8      this.lastName = lastName;
9      this.name = firstName + " " + lastName;
10   }
11
12   public String getFirstName () {
13     return firstName;
14   }
15   public String getLastName () {
16     return lastName;
17   }
18   public String getName () {
19     return name;
20   }
21
22   public Object readResolve() {
23     return new Person (firstName, lastName);
24   }
25 }
```

Der erste Schritt dazu ist, das Feld mit dem Schlüsselwort *transient* zu versehen. Dadurch wird es vom Serialisierungsmechanismus ignoriert. In einem zweiten Schritt müssen wir dafür sorgen, dass es beim Deserialisieren initialisiert wird. Dazu ruft *readResolve()* den Konstruktor auf und gibt eine neue, voll initialisierte *Person*-Instanz zurück.

Noch mehr Kontrolle bietet das Implementieren der Methoden *readObject()* und *writeObject()*, die das Lesen und Schreiben des Objektzustands direkt definieren (Listing 8). Das ist zum Beispiel nützlich, um große, komplexe Objekte kompakt zu repräsentieren – der Code im Listing zeigt das API, ohne in diesem Fall einen Mehrwert zu bringen.

Listing 8

```
1  public class Person implements Serializable {
2    private String firstName;
3    private String lastName;
4
5    public Person (String firstName, String lastName) {
6      this.firstName = firstName;
7      this.lastName = lastName;
8    }
9
10   public String getFirstName () {
11     return firstName;
12   }
13   public String getLastName () {
14     return lastName;
15   }
16
17   private void writeObject(ObjectOutputStream s) throws IOEx
18     s.writeUTF (firstName);
19     s.writeUTF (lastName);
20   }
21
22   private void readObject(ObjectInputStream s) throws IOExce
23     firstName = s.readUTF ();
24     lastName = s.readUTF ();
25   }
26 }
```

Zum Schreiben einzelner Werte hat *ObjectOutputStream* Methoden wie *writeInt()*, *writeBoolean()* oder *writeUTF()* – *ObjectInputStream* hat entsprechende Methoden zum Lesen.

Versionierung – muss sein!

Serialisierung und Deserialisierung von Objekten können in unterschiedlichen JVMs erfolgen, die potenziell auf unterschiedlichen Computern laufen. Und es kann z. B. bei Dateien beliebig viel Zeit zwischen dem Schreiben und Lesen liegen. So kann es passieren, dass Serialisieren und Deserialisieren auf verschiedenen Softwareständen erfolgt. Das kann zu Problemen führen.

Zunächst müssen alle benötigten Klassen beim Deserialisieren im Classpath liegen, sonst wirft der *ObjectInputStream* eine *ClassNotFoundException*. Man kann also durch Serialisieren nicht einfach Code übertragen, der dann auf einer anderen Maschine

ausgeführt wird, und das ist gut so – alles andere wäre eine riesige Sicherheitslücke. Für eine Beschreibung, wie eine ungeschickte Implementierung der Deserialisierung z. B. in *commons-collections* trotzdem die Tür für Remote Code Execution öffnet, siehe [1].

Außerdem müssen die Klassen beim Schreiben zu den Klassen beim Lesen passen – wenn beim Schreiben eine Person zum Beispiel nur einen String mit dem gesamten Namen hat, beim Lesen aber eine spätere Version der *Person*-Klasse getrennte Vor- und Nachnamen hat, wirft der *ObjectInputStream* ebenfalls eine Exception.

Als zusätzliche Sicherheit schreibt der Serialisierungsmechanismus für jede verwendete Klasse noch einen Long mit einer Versionsnummer der Klasse, der *serialVersionUID*. Per Default ermittelt Java diese Zahl als eine Art Hashwert über die Definition der Klasse, sodass Änderungen an der Klasse den Wert ändern. Der *ObjectInputStream* überprüft dann beim Lesen, ob die Klasse noch dieselbe *serialVersionUID* hat wie beim Schreiben.

Der Algorithmus zum automatischen Ermitteln der *serialVersionUID* ist sehr konservativ, und manche Änderungen am Code ändern ihren Wert, obwohl das serialisierte Format kompatibel bleibt. Deshalb empfiehlt sogar die Javadoc von *Serializable*, dass jede serialisierbare Klasse eine explizite Version definieren sollte. Dazu dient ein Feld *static final long serialVersionUID = …;* mit beliebiger Sichtbarkeit.

Java-Serialisierung: Wann passt sie, wann nicht?

Java-Serialisierung ist ein einfacher Mechanismus, um Objektstrukturen zu speichern oder über Netzwerke zu übertragen. Er bietet über die Default-Mechanismen hinaus Möglichkeiten, die konkrete Repräsentation einzelner Datentypen festzulegen, z. B. um sie effizienter zu gestalten. Serialisierung ist weit verbreitet und hat ihre Stärken im Umgang mit Objektidentität und Objektnetzen. Wenn effizientes Speichern oder Übertragen von Objekten im Vordergrund steht, gibt es aber kompaktere und schnellere Alternativen zur Java-Serialisierung. Insbesondere ProtoBuf von Google (http://bit.ly/1mISy49) ist weit verbreitet, extrem effizient und für verschiedene Programmiersprachen verfügbar.

Java-Trickkiste

Ungewöhnliche Bibliotheken, skurrile Codeschnipsel, idiomatische Leckerbissen und der Blick hinter die Kulissen der verbreitetsten Programmiersprache der Welt. Das ist das Thema dieser Kolumne von **Arno Haase**, die sowohl für Neulinge als auch alte Hasen Wissenswertes bereithält.

Bisher erschienen:

- Aus der Java-Trickkiste: Java-Serialisierung – wann passt sie, wann nicht? (https://jaxenter.de/aus-der-java-trickkiste-java-serialisierung-wann-passt-sie-wann-nicht-35558)
- Speicher und Garbage Collection (https://jaxenter.de/aus-der-java-trickkiste-speicher-und-garbage-collection-31574)
- Seiteneffektfreier Code – Mehrere Threads (https://jaxenter.de/aus-der-java-trickkiste-seiteneffektfreier-code-mehrere-threads-30655)
- Listen ohne Seiteneffekte (https://jaxenter.de/aus-der-java-trickkiste-listen-ohne-seiteneffekte-27892)
- Code ohne Seiteneffekte (https://jaxenter.de/kolumne-java-trickkiste-25200)
- Microbenchmarking (https://jaxenter.de/aus-der-java-trickkiste-microbenchmarking-24155)
- Class Loading Reloaded (https://jaxenter.de/aus-der-java-trickkiste-class-loading-reloaded-20317)
- Class Loading (https://jaxenter.de/aus-der-java-trickkiste-class-loading-20271)
- Der JIT-Compiler von Hotspot (https://jaxenter.de/java-trickkiste-der-jit-compiler-von-hotspot-19878)
- Hotspot-Schalter (https://jaxenter.de/aus-der-java-trickkiste-hotspot-schalter-19041)
- Bytecode-Analyse im Eigenbau (https://jaxenter.de/bytecode-analyse-im-eigenbau-17102)
- Classpath-Scan im Eigenbau (https://jaxenter.de/classpath-scan-im-eigenbau-aus-der-java-trickkiste-12963)
- Performancemythen (https://jaxenter.de/aus-der-java-trickkiste-performancemythen-103)
- Patterns zum Instanziieren von Klassen (https://jaxenter.de/java-trickkiste-patterns-zum-instanziieren-von-klassen-377)
- Pleiten, Pech und Pannen mit Threads (https://jaxenter.de/pleiten-pech-und-pannen-mit-threads-java-trickkiste-408)
- Nebenläufig – und auch schnell? Concurrency mit und ohne Locks (https://jaxenter.de/nebenlaufig-und-auch-schnell-concurrency-mit-und-ohne-locks-java-trickkiste-442)
- Referenzen mit Sonderstatus (https://jaxenter.de/aus-der-java-trickkiste-referenzen-mit-sonderstatus-510)
- JDBC-Treiber selbstgebaut (https://jaxenter.de/jdbc-treiber-selbstgebaut-java-trickkiste-636)
- Überraschende Effekte mit Java Reflection (https://jaxenter.de/uberraschende-effekte-mit-java-reflection-1013)
- Die Tücken bei der Performancemessung (https://jaxenter.de/die-tucken-bei-der-performancemessung-java-trickkiste-1079)

- Von Checked und Unchecked Exceptions (https://jaxenter.de/java-trickkiste-von-checked-und-unchecked-exceptions-1350)

Links & Literatur

[1] Kaiser, Matthias: „Exploiting Deserialization Vulnerabilities in Java“ (http://bit.ly/1GKzjIl)

GESCHRIEBEN VON

Arno Haase

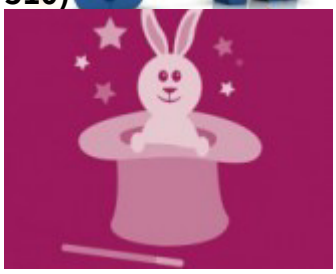
Arno Haase ist freiberuflicher Softwareentwickler. Er programmiert Java aus Leidenschaft, arbeitet aber auch als Architekt, Coach und Berater. Seine Schwerpunkte sind modellgetriebene Softwareentwicklung, Persistenzlösungen mit oder ohne relationaler Datenbank und nebenläufige und verteilte Systeme. Arno spricht regelmäßig auf Konferenzen und ist Autor von Fachartikeln und Büchern. Er lebt mit seiner Frau und seinen drei Kindern in Braunschweig.

Alle Beiträge von Arno Haase (https://jaxenter.de/author/arnohaase)

DAS KÖNNTE SIE AUCH INTERESSIEREN!

Java-Trickkiste: Der JIT-Compiler von Hotspot (https://jaxenter.de/java-trickkiste-der-jit-compiler-von-hotspot-19878)

Aus der Java-Trickkiste: Referenzen mit Sonderstatus
(https://jaxenter.de/aus-der-java-trickkiste-referenzen-mit-sonderstatus-510)



JDBC-Treiber selbstgebaut [Java-Trickkiste]
(https://jaxenter.de/jdbc-treiber-selbstgebaut-java-trickkiste-636)

0

(https://www.facebook.com/sharer/sharer.php?u=https://jaxenter.de/aus-der-java-trickkiste-java-serialisierung-wann-passt-sie-wann-nicht-35558)
(https://twitter.com/home?status=https://jaxenter.de/aus-der-java-trickkiste-java-serialisierung-wann-passt-sie-wann-nicht-35558)
(https://plus.google.com/share?url=https://jaxenter.de/aus-der-java-trickkiste-java-serialisierung-wann-passt-sie-wann-nicht-35558)
(https://www.linkedin.com/shareArticle?mini=true&url=https://jaxenter.de/aus-der-java-trickkiste-java-serialisierung-wann-passt-sie-wann-nicht-35558)

KOMMENTARE

Rüdiger Möller 2016-03-13 12:24:49

Kleiner Hinweis:

mit fst (https://github.com/RuedigerMoeller/fast-serialization) gibt es eine kompatible (drop-in) implementierung, die um Faktoren (4 bis >10) mal schneller ist.

Gerade in verteilten Systemen wird die doch recht gemächliche JDK Implementierung schnell ein Flaschenhals. Alternativen wie Protobuf sind in der Regel deutlich aufwendiger und in der Entwicklung deutlich teurer (eigene Datentypen, Generate, kein Objectgraph support etc., Wandlung DatenNachrichten), sodaß die Nutzung von Serialisierung gerade in relativ homogenen verteilten Systemen einen signifikanten Kostenvorteil bringt.

Schreibe einen Kommentar

Deine E-Mail-Adresse wird nicht veröffentlicht.

Name

E-Mail

Website

Kommentar abschicken

POLLS

Welche Java-Version setzen Sie ein?

- ☐ Java 8
- ☐ Java 7
- ☐ Java 6
- ☐ Java 5
- ☐ Java 1.4
- ☐ Java 1.3
- ☐ Java 1.2
- ☐ Java 1.1
- ☐ Java 1.0

✔ VOTE

(https://jaxenter.de/magazine/eclipse-magazin)

KOLUMNEN

(https://jaxenter.de/tag/kolumne)

Angular 4.3.0 erschienen: Die neuen Features auf einen Blick (https://jaxenter.de/angular-4-3-0-erschienen-59537)

DevOps-Stories: So wird hier gearbeitet! (https://jaxenter.de/devops-stories-teamvertrag-59168)

Web-UIs mit Java erstellen: So geht's mit Vaadin im Core Servlet Container (https://jaxenter.de/vaadin-undertow-java-servlet-59394)

EnterpriseTales: Auf dem Weg zur SPA (https://jaxenter.de/enterprisetales-auf-dem-weg-zur-spa-58941)

Women in Tech – „Macht einen Master in Selbstdarstellung“ (https://jaxenter.de/women-in-tech-saskia-kurz-58866)

LETZTE KOMMENTARE

Sebastian Hauptsächlich (Dark) Ambient (z.B. viele Cryo Chamber-Sachen), gelegentlich auch Post-Rock (z.B. neuere Solstafir). Ruhigere Videospiel-Soundtracks (z.B. Hyper Light Drifter) oder atmosphärischere Black Metal-Sachen (z.B. Paysage...

Bits & Beats: Welche Musik Entwickler beim Programmieren hören
(https://jaxenter.de/musik-entwickler-programmieren-59501#comment-171900)

Det Musik ohne Gesang (Lindsey Sterling). Vor allem deutsche Texte geht gar nicht. Am besten Ambient oder Lounge, wegen der Kontinuität. Auch gut: Soundtrack des Planetariums...

Bits & Beats: Welche Musik Entwickler beim Programmieren hören
(https://jaxenter.de/musik-entwickler-programmieren-59501#comment-171877)

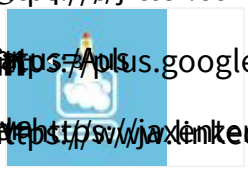
Stephan Zerhusen Animals as Leaders sind der Hammer! Generell geht Progressive Metal gut, aber auch Drum'n'Bass.

Bits & Beats: Welche Musik Entwickler beim Programmieren hören
(https://jaxenter.de/musik-entwickler-programmieren-59501#comment-171866)

FEATURED POSTS

Angular 4.3.0 erschienen: Die neuen Features auf einen Blick
(https://jaxenter.de/angular-4-3-0-erschienen-59537)

Bits & Beats: Welche Musik Entwickler beim Programmieren hören
(https://jaxenter.de/musik-entwickler-programmieren-59501)



Spring Boot Tutorial: In 10 Schritten zur Microservices-Architektur
(https://jaxenter.de/spring-boot-tutorial-microservices-cloud-foundry-kubernetes-58695)

Funktional(er) mit Java programmieren: Vavr – der Java-Slang
(https://jaxenter.de/vavr-javaslang-funktional-programmieren-59108)

NEWSLETTER

Alle Neuigkeiten die wichtig sind, mit dem JAXenter Newsletter! (https://jaxenter.de/newsletter)

Jetzt bestellen! (https://jaxenter.de/newsletter)

JAX 2017

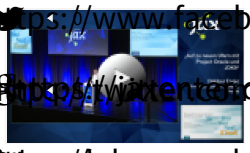
Ab durch die Schallmauer: Mit Angular hochperformante Anwendungen erstellen (https://jaxenter.de/angular-performance-steyer-jax-59479)

JavaFX, Swing oder HTML5: Quo vadis Java-Desktop? (https://jaxenter.de/javafx-swing-html5-java-desktop-59382)

Tutorials für Spring Boot, Java 9 & Drohnensteuerung – unsere Top-Themen der Woche (https://jaxenter.de/spring-boot-java-9-drohne-59326)

10 Take-aways von der JAX 2017: Microservices, Container, Java 9 & More (https://jaxenter.de/10-take-aways-jax-2017-59056)

Erfolgreiche agile Projekte: Wie man ein ganzes Unternehmen transformiert (https://jaxenter.de/agile-softwareentwicklung-schmidt-59098)



Java 9, Project Jigsaw & JDK 9: So lässt sich die Arbeit beschleunigen und verbessern (https://jaxenter.de/java-9-jdk-9-jigsaw-keynote-59010)

Diversität in der Tech-Branche: "Diverse Teams treffen bessere Entscheidungen" (https://jaxenter.de/diversitaet-interview-tracy-miranda-58630)

Java 9 für Library-Entwickler: "Die Unsafe-Klasse sollte man einfach nicht mehr nutzen" (https://jaxenter.de/java-9-interview-uwe-schindler-58580)

KONFERENZEN

JAX, W-JAX (https://jax.de?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
JAX London (https://jaxlondon.com?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
JAX DevOps (London) (https://devops.jaxlondon.com/?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
JAX Finance (London) (https://finance.jaxlondon.com/?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
Business Technology Days (https://btdays.de?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
BigData Conference (http://bigdatacon.de?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
DevOps Conference (https://devopsconference.de/?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
IoT Conference (https://iotcon.de?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
WebTech Conference (https://webtechcon.de?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
MobileTech Conference (https://mobiletechcon.de?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
BASTA! (https://basta.net?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
International PHP Conference (https://phpconference.com?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
International JavaScript Conference (https://javascript-conference.com/?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
API Conference (https://apiconference.de/?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
Webinale (https://webinale.de/de/?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)
ENTER (https://entwickler-konferenz.de/de/?utm_source=jaxenter.de&utm_medium=referral&utm_term=general&utm_campaign=footer)

(https://www.youtube.com/user/VideosJAXenter)

S&S MEDIA

entwickler.kiosk (http://entwickler-kiosk.de)
Entwickler Akademie (http://entwickler-akademie.de)
entwickler press (http://entwickler-press.de)
Über Software & Support Media (http://sandsmedia.com/de/unternehmen)
Sponsoring & Advertising (http://sandsmedia.com/de/sponsoring-advertising)
Karriere & Stellenangebote (http://sandsmedia.com/de/karriere)
Kontakt & Anschrift (http://sandsmedia.com/de/kontakt)

FOLLOW US

(https://www.facebook.com/JAXenter)
(https://twitter.com/jaxenter)
(http://vimeo.com/jaxtv)
(https://jaxenter.de/feed)

NEWSLETTER

Jetzt den JAXenter Newsletter bestellen! (https://jaxenter.de/newsletter)

(https://jaxenter.de/newsletter)

(https://www.facebook.com/sharer/sharer.php?u=https://jaxenter.de/aus-der-java-trickkiste-java-serialisierung-wann-passt-sie-wann-nicht-35558)
(https://plus.google.com/share?url=https://jaxenter.de/aus-der-java-trickkiste-java-serialisierung-wann-passt-sie-wann-nicht-35558)
(http://www.linkedin.com/shareArticle?mini=true&url=https://jaxenter.de/aus-der-java-trickkiste-java-serialisierung-wann-passt-sie-wann-nicht-35558)

Software & Support Media Group

(http://sandsmedia.com/)

Datenschutz (https://jaxenter.de/datenschutz)

0

Impressum (https://jaxenter.de/impressum)

(https://www.facebook.com/sharer/sharer.php?u=https://jaxenter.de/aus-der-java-trickkiste-java-serialisierung-wann-passt-sie-wann-nicht-35558)

(https://twitter.com/home?status=Aus%20der%20Java-Trickkiste:%20Java-Serialisierung%20-%20wann%20passt%20sie,%20wann%20nicht?%20https://jaxenter.de/aus-der-java-trickkiste-java-serialisierung-wann-passt-sie-wann-nicht-35558)

(https://plus.google.com/share?url=https://jaxenter.de/aus-der-java-trickkiste-java-serialisierung-wann-passt-sie-wann-nicht-35558)

(https://www.linkedin.com/shareArticle?mini=true&url=https://jaxenter.de/aus-der-java-trickkiste-java-serialisierung-wann-passt-sie-wann-nicht-35558)