Ausgabe 06/2019

Java aktuell

iJUG Verbund
www.ijug.eu

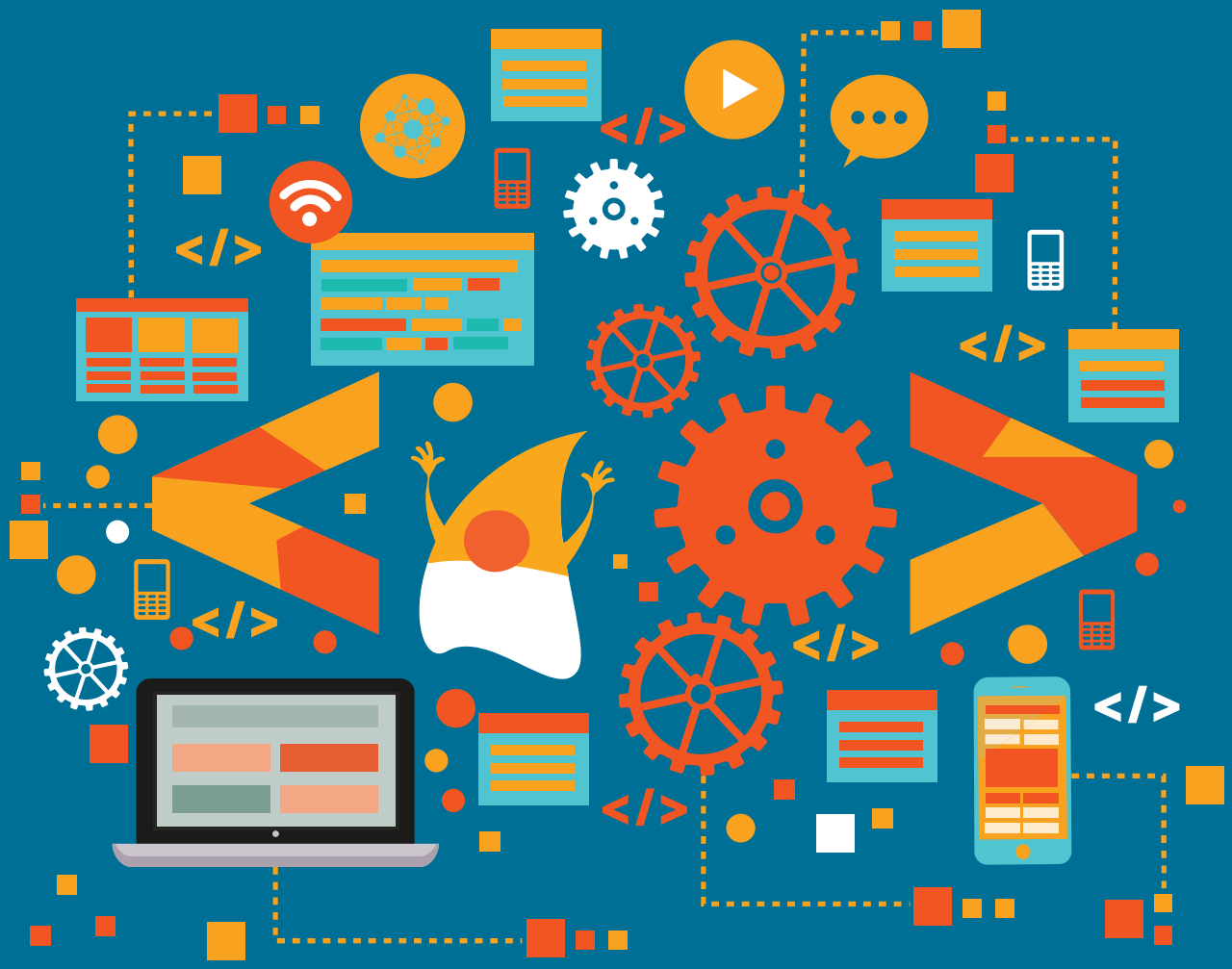
JDK 13
Was gibt's Neues?

GitOps
Mit Helm und Kubernetes durchstarten

Microservices
Wie sinnvoll sind sie bei Transaktionen?

Wie sauberer Code Ihr Projekt nachhaltig verbessert

PROJEKT-BOOSTER <CLEAN CODE>



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 06

virtual7

WIR GESTALTEN DIE DIGITALE ZUKUNFT DEUTSCHLANDS.

Werde Teil unseres Expertenteams und finde Deinen Einstieg als Developer oder Consultant in den Bereichen:

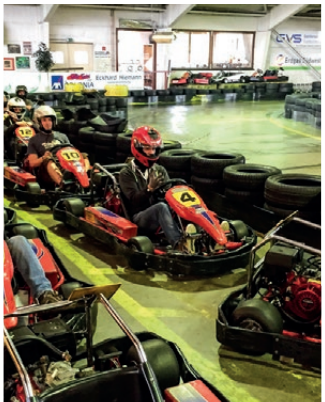
- Business Intelligence
- Data Analytics
- Middleware
- Infrastructure & Operations

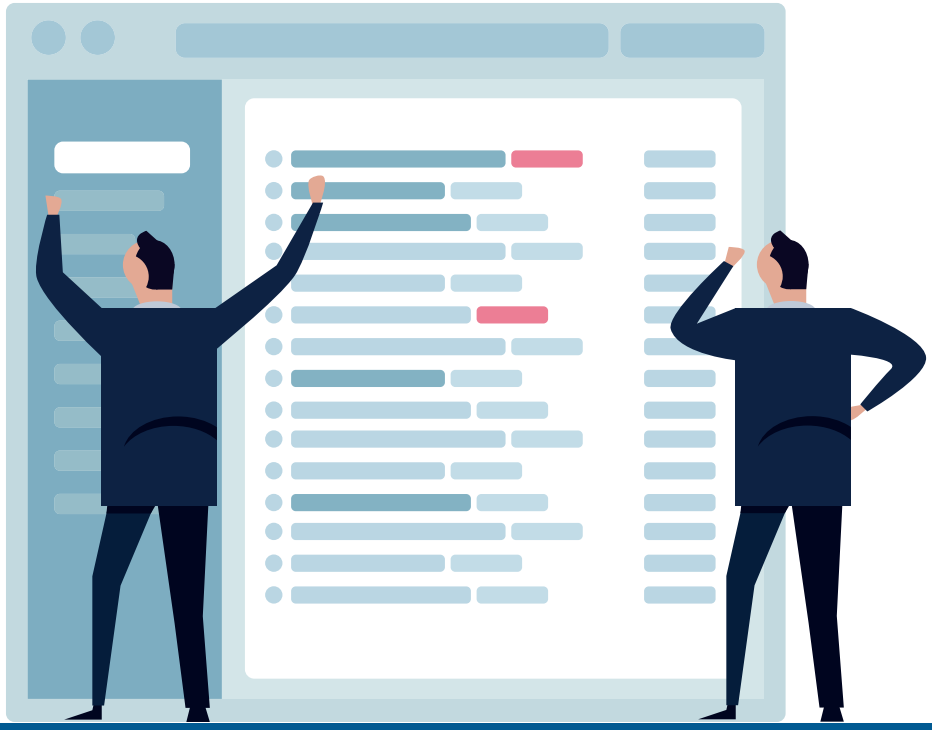
WIR SUCHEN DICH!

Mehr Informationen: karriere.virtual7.de





Unbekannte Kostbarkeiten des SDK Heute: Die Klasse „Files“

Bernd Müller, Ostfalia

Das Java SDK enthält eine Reihe von Features, die wenig bekannt sind. Wären sie bekannt und würden sie verwendet, könnten Entwickler viel Arbeit und manchmal sogar zusätzliche Frameworks einsparen. Wir wollen in dieser Reihe derartige Features des SDK vorstellen: die unbekannten Kostbarkeiten.

Mit Java 7 wurde das Package `java.nio.file` eingeführt, das unter anderem die Klasse `Files` enthält. Mit Java 8 wurden dieser Klasse einige neue Methoden hinzugefügt, weitere mit Java 11 und 12. Diesen neuen Methoden gilt die Aufmerksamkeit unserer heutigen unbekannten Kostbarkeiten.

Java IO reloaded, reloaded, reloaded...

Eine Programmiersprache, die in der Praxis eingesetzt werden soll, muss Funktionalitäten zur Ein- und Ausgabe insbesondere auch in Bezug zum Dateisystem bereitstellen. Man möchte fast behaupten: je mehr, desto besser. Leser im Alter des Autors können sich eventuell noch an Pascal *[1]* (nicht Turbo-Pascal) und Modula *[2]* von Niklaus Wirth erinnern. Diese Sprachen waren mit sehr spartanischen I/O-APIs ausgestattet, die wahrscheinlich dazu beigetragen haben, dass sie sich nicht in der Praxis durchsetzen konnten. Java besaß von Anfang an eine reichhaltige Bibliothek von I/O-Klassen, zur Zeit von Java 1 im Package `java.io`. Java entwickelte sich weiter, auch wenn manche das Gegenteil behaupteten. Mit Java 1.4 wurde 2002 Java NIO (New I/O) im Package `java.nio` eingeführt, sogar in einem eigenen JSR, dem JSR 51: *New I/O for the Java Platform.* Diesem API

```
• public static BufferedReader newBufferedReader(Path path)
• public static BufferedWriter newBufferedWriter(Path path,OpenOption... options)
• public static List<String> readAllLines(Path path)
• public static Path write(Path path,Iterable<? extends CharSequence> lines,OpenOption... options)
• public static Stream<Path> list(Path dir)
• public static Stream<Path> walk(Path start,int maxDepth,FileVisitOption... options)
• public static Stream<Path> walk(Path start,FileVisitOption... options)
• public static Stream<Path> find(Path start,int maxDepth,BiPredicate<Path,BasicFileAttributes>matcher,
                                 FileVisitOption... options)
• public static Stream<String> lines(Path path,Charset cs)
• public static Stream<String> lines(Path path)
```

Listing 1

```
public void listFilesWithExtension(String extension) {
  try (Stream<Path> walk =
       Files.walk(Paths.get("").toAbsolutePath())) {
    walk.map(f -> f.toString())
        .filter(f -> f.endsWith(extension))
        .forEach(System.out::println);
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

Listing 2

	Java	find	Java GraalVM
real	1,076 s	0,609 s	0,836 s
user	2,344 s	0,261 s	0,290 s
sys	0,732 s	0,344 s	0,541 s

Tabelle 1

hen, dass Java fast doppelt so lange benötigt wie der „find"-Befehl. Die tatsächliche CPU-Benutzung beträgt fast das Zehnfache. Das Executable, das mit der GraalVM durch den Befehl `native-image` erzeugt wurde, entspricht in etwa dem Linux-„find"-Befehl.

Java 11 und 12

Wie bereits angedeutet, entwickelt sich Java mit hoher Geschwindigkeit weiter. Das Einlesen eines Dateiinhalts in einen String beziehungsweise das Schreiben eines Strings in eine Datei sind seit Java 11 mit einem einzigen Methodenaufruf möglich. Das Suchen nach der (ersten) Stelle, an der sich zwei Dateien unterscheiden, gibt es seit Java 12. Die Signaturen der entsprechenden Methoden zeigt *Listing 3*.

Zusammenfassung

Seit Java 7 gibt es die Klasse `Files`, die im Rahmen von NIO.2 entstanden ist. Mit Java 8, 11 und 12 kamen eine Reihe weiterer Methoden hinzu, die verschiedene Aufgaben mit I/O stark vereinfachen.

```
• public static String readString(Path path)
• public static String readString(Path path, Charset cs)
• public static Path writeString(Path path, CharSequence csq, OpenOption... options)
• public static Path writeString(Path path, CharSequence csq, Charset cs, OpenOption... options)
• public static long mismatch(Path path, Path path2)
```

Listing 3

hat Ron Hitchens ein ganzes Buch gewidmet *[3]*. Es umfasst knapp doppelt so viele Seiten wie die Bücher über Pascal und Modula. Mit Java 7 wurden 2011 abermals mit einem JSR, nun dem JSR 203, *More New I/O APIs for the Java Platform („NIO.2")*, die I/O-Funktionalitäten von Java erweitert. Wir gehen davon aus, dass dem Leser dies mehr oder weniger bekannt ist, und konzentrieren uns darauf, was danach geschah. Mit Java 8 wurden Lambda-Ausdrücke und Streams eingeführt, sodass die zehn neuen Methoden der Klasse `Files` eventuell etwas ins Hintertreffen geraten sind. *Listing 1* zeigt eine Übersicht dieser Methoden.

Wir belassen es bei der Angabe der Signaturen, um den Umfang des Artikels nicht zu sprengen, und entwickeln nur ein kleines Beispiel mit der zweiten `walk()`-Methode. Beide `walk()`-Methoden liefern einen Stream zurück und arbeiten daher "lazy". Das Beispiel, alle Dateien mit einer bestimmten Dateinamenserweiterung auszugeben, kann daher wie in *Listing 2* zu sehen realisiert werden.

Die `walk()`-Methode durchsucht den Verzeichnisbaum depth-first und ist durch die Stream-inherente Lazy-Eigenschaft auch für große Verzeichnisbäume geeignet.

Nach unserer Meinung ist dies eine sehr elegante Lösung. Es stellt sich aber die Frage, was uns diese Eleganz in Bezug auf die Laufzeit kostet. Wir haben dies gemessen, wenngleich nicht unter wirklichen Benchmark-Bedingungen. *Tabelle 1* gibt diese Messung wieder. Die Zeiten wurden mit dem Unix-Befehl `time` ermittelt.

Als Unix-Befehl wurde `find . -name *.java` verwendet. Die Ausgaben der drei Alternativen wurden auf `/dev/null` umgeleitet. Wir sehen, dass Java fast doppelt so lange benötigt wie der „find"-Befehl. Die tatsächliche CPU-Benutzung beträgt fast das Zehnfache. Das Executable, das mit der GraalVM durch den Befehl `native-image` erzeugt wurde, entspricht in etwa dem Linux-„find"-Befehl.

Insbesondere das Lesen einer Datei in einen String beziehungsweise das Schreiben eines Strings in eine Datei mit einem einzigen Befehl sollte in das Repertoire eines Java-Entwicklers gehören.

Referenzen

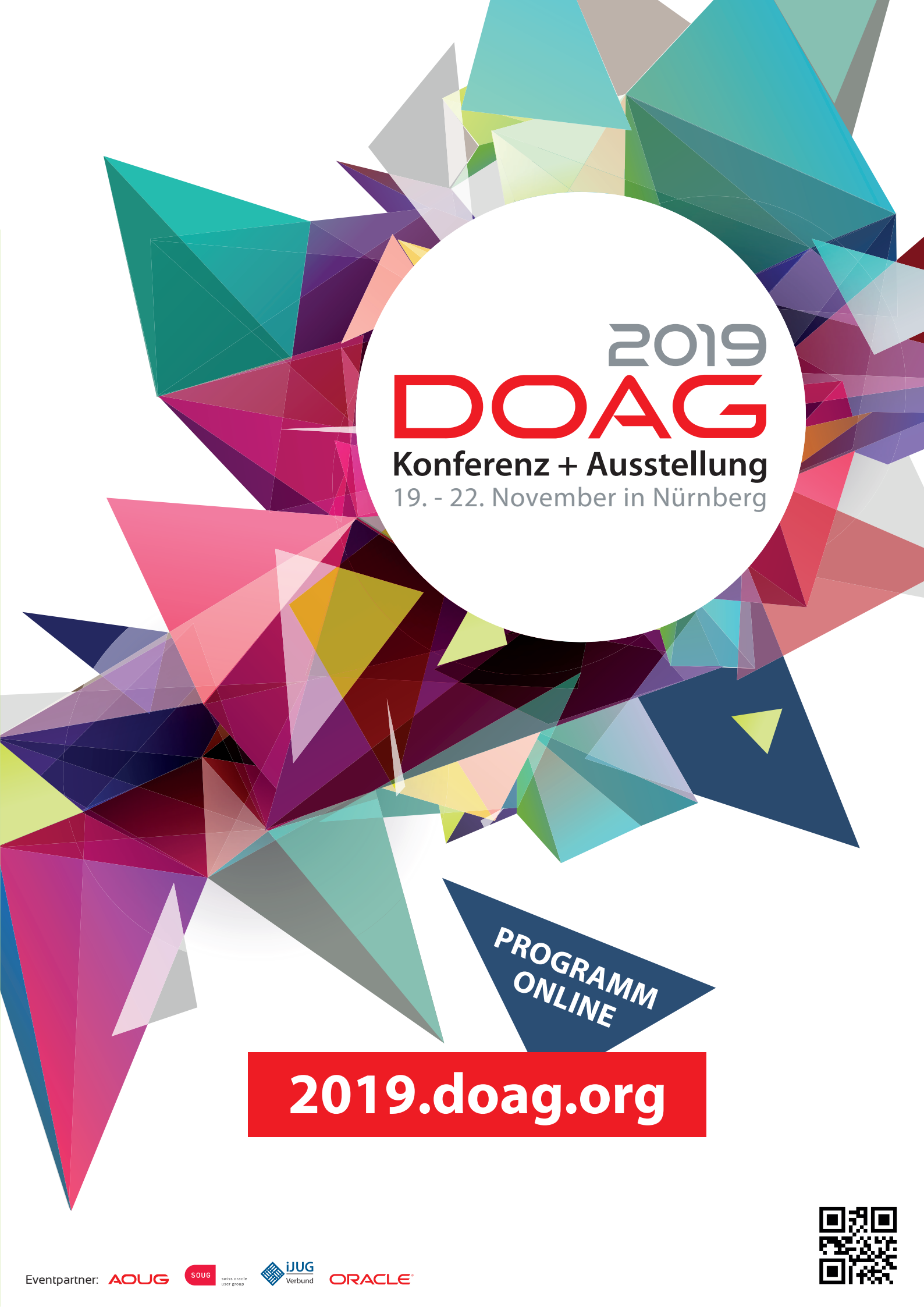
[1] Kathleen Jensen, Niklaus Wirth. PASCAL User Manual and Report, Springer-Verlag, 1975.

[2] Niklaus Wirth. Programming in MODULA-2, Springer-Verlag, 1982.

[3] Ron Hitchens. Java NIO, O'Reilly, 2002.

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.


2019
DOAG
Konferenz + Ausstellung
19. - 22. November in Nürnberg
PROGRAMM ONLINE
2019.doag.org
Eventpartner:
AOUG
SOUG
swiss oracle user group
iJUG Verbund
ORACLE