# Java aktuell

iJUG Verbund
www.ijug.eu

**Newcomer-Beiträge**
AWS Serverless Lambda-Funktionen, Projektkommunikation, Mob Programming, Einstieg in die Softwareentwicklung

**Spring Native**
Java und Spring Boot für die Cloud, Testen von Kubernetes-Controllern und -Operatoren, Strings

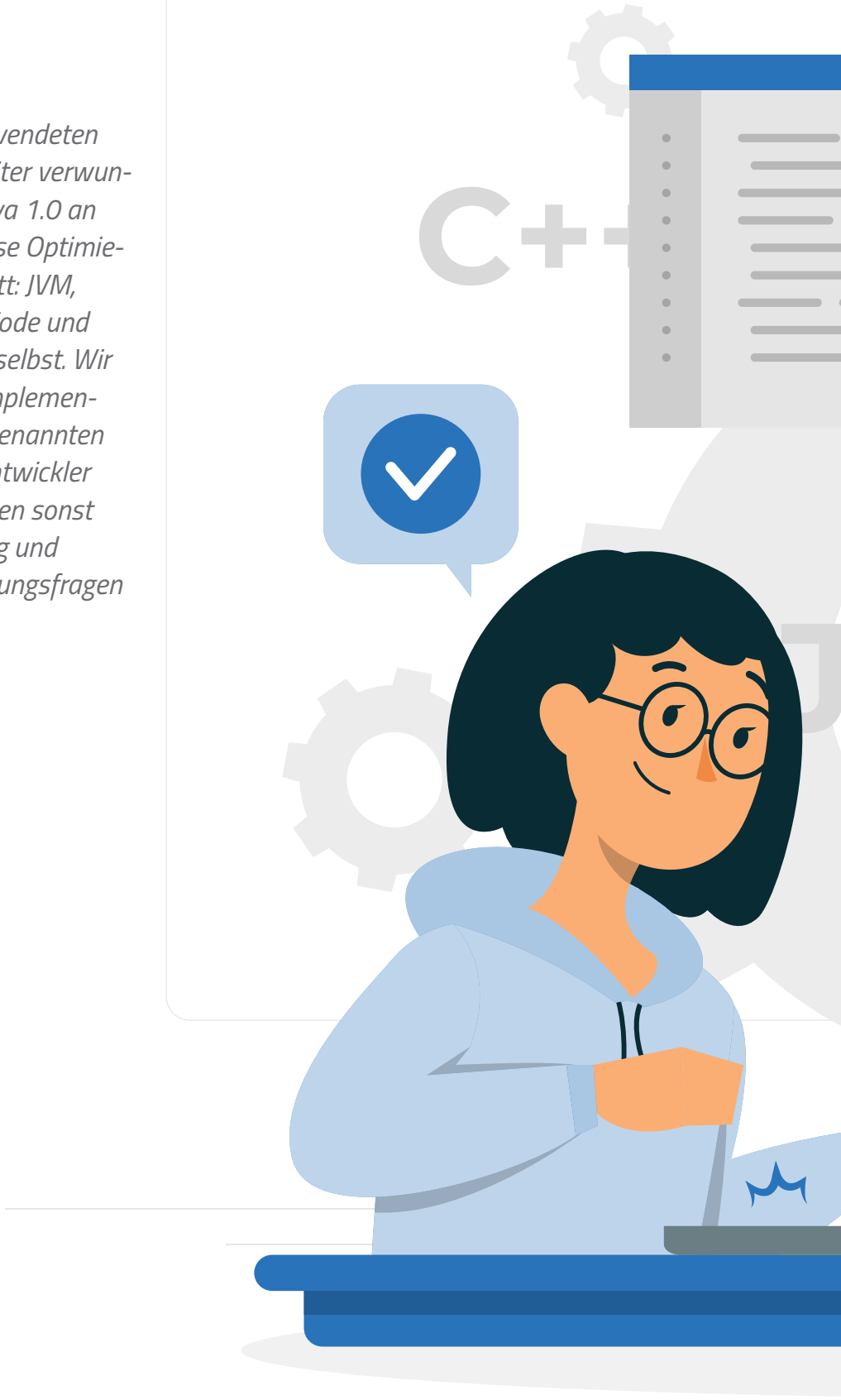
## Next Generation

Ausgabe 03/2023
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
4 191978 304903 03

# Was jeder Java-Entwickler über Strings wissen sollte

**Bernd Müller, Ostfalia**

*Strings sind die am häufigsten verwendeten Java-Objekte. Es ist daher nicht weiter verwunderlich, dass JDK-Entwickler seit Java 1.0 an String-Optimierungen arbeiten. Diese Optimierungen finden auf allen Ebenen statt: JVM, Garbage Collector, Compiler, Byte-Code und natürlich auch in der String-Klasse selbst. Wir werfen einen kleinen Blick in die Implementierungsdetails von Strings in den genannten Bereichen und hoffen, dass Java-Entwickler nach der Lektüre einen Einblick in den sonst so unspektakulären Datentyp String und zusammenhängende Implementierungsfragen bekommen haben.*




HTML
CSS
Java

## Einordnung

Strings sind die am häufigsten verwendeten Java-Objekte. Beginnend bei Exception-Messages über Log-Nachrichten, Beschriftungen von UI-Elementen und Datenaustauschformaten wie XML und JSON, basieren alle genannten Punkte und noch viele mehr auf Strings. Durch die allgegenwärtige Verwendung von Strings ist deren speichereffiziente Repräsentation sowie laufzeiteffiziente Verarbeitung von zentraler Bedeutung für die Leistungsfähigkeit einer Sprache beziehungsweise Plattform. Dies gilt sowohl für Java als auch für alle anderen Programmiersprachen. Bereits in der initialen Version 1.0 von Java wurden daher entsprechende Optimierungen realisiert, die in den letzten 25 und mehr Jahren meist in der Form von JEPs *[1]* weiter vervollständigt wurden und immer noch werden. Wir stellen im Folgenden vor, wie Strings intern gespeichert wurden und werden, wie Duplikate gefunden und entfernt sowie Strings konkateniert werden.

## Der String-Pool

Nach Abschnitt 3.10.5 String Literals der Java-Sprachdefinition gibt es String-Literale nur einmal in der JVM:

*„Moreover, a string literal always refers to the same instance of class String. This is because string literals – or, more generally, strings that are the values of constant expressions – are ‚interned' so as to share unique instances, as if by execution of the method String.intern."*

```
public class PoolTest {

  String str1 = "Hello, World!";
  String str2 = "Hello, World!";
  String str3 = new String("Hello, World!");

  @Test
  public void same() {
     Assertions.assertSame(str1, str2);
  }

  @Test
  public void same2() {
     Assertions.assertSame("Hel" + "lo", "Hel" + "lo");
  }

  @Test
  public void same3() {
     Assertions.assertSame("Hello, World!", "Hello, World!");
  }

  @Test
  public void notSame() {
     Assertions.assertNotSame(str1 + str1, str2 + str2);
  }

  @Test
  public void sameAfterInterning() {
     Assertions.assertSame((str1 + str1).intern(), (str2 + str2).intern());
  }

  @Test
  public void equals() {
     Assertions.assertEquals(str1 + str1, str2 + str2);
  }

  @Test
  public void notSameNewConstructed() {
     Assertions.assertNotSame(str1, str3);
  }

}
```

*Listing 1*

Aber nicht nur String-Literale wie „`Hello`", sondern auch konstante Ausdrücke von String-Literalen wie etwa „`Hello" + " World! "`, die schon vom Compiler zusammengefasst werden, existieren in der JVM nur einmal. Unabhängig von einer eventuell tausendfachen Verwendung in Hunderten verschiedener Klassen sind die String-Literale „`Hello`" und „`Hello World! `" also Singletons in der JVM. Diese Eindeutigkeit wird durch eine weitere Forderung der Sprachdefinition im Abschnitt *12.5 Creation of New Class Instances* gewährleistet:

*„Loading of a class or interface that contains a string literal or a text block may create a new String object to denote the string represented by the string literal or text block. (This object creation will not occur if an instance of String denoting the same sequence of Unicode code points as the string represented by the string literal or text block has previously been interned.)"*

Was es mit einem Unicode-Code-Point auf sich hat, sehen wir später. Was ein „internter" String ist (wir bleiben bei der englischen Version, da das deutsche Internieren negativ konnotiert ist), entnehmen wir dem JavaDoc der Methode `String.intern()`:

*„Returns a canonical representation for the string object. A pool of strings, initially empty, is maintained privately by the class String. When the intern method is invoked, if the pool already contains a string equal to this String object as determined by the equals(Object) method, then the string from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned. It follows that for any two strings s and t, s.intern() == t.intern() is true if and only if s.equals(t) is true. All literal strings and string-valued constant expressions are interned. String literals are defined in section 3.10.5 of the The Java Language Specification."*

Der genannte String-Pool ist eine in C implementierte Hash-Table im nativen Speicher der JVM, die Strings selbst liegen im Heap. Die C-Implementierung erlaubt das von Java-Hash-Tables gewohnte dynamische Wachstum nicht, die Anzahl der Buckets ist fest. Seit Java 11 ist die Größe 65536, obwohl die gängige Literatur zu einer Primzahl rät. Falls dieser Wert nicht optimal ist und beispielsweise viele Kollisionen auftreten, kann die Größe beim Start der JVM mit der Option `-XX:StringTableSize=N` eingestellt werden. Wie die Auslastung der Hash-Table tatsächlich ist, kann durch die JVM-Option `-XX:PrintStringTableStatistics` erfragt werden, die zur Ausgabe der Statistik des String-Pools führt, wenn die JVM beendet wird.

Das Verfahren, wie Java die Singleton-Eigenschaft von String-Literalen garantiert, kann nun wie folgt zusammengefasst werden. Der Compiler erzeugt aus Ausdrücken,

die nur aus String-Literalen bestehen, den konkatenierten String bereits für den Byte-Code. Die im Byte-Code vorhandenen String-Literale werden beim Laden der Klasse in die JVM dahingehend geprüft, ob sie bereits im String-Pool vorhanden sind. Falls ja, wird die Referenz auf den String zurückgegeben, der String aber nicht nochmals in den Pool aufgenommen. Falls das Literal noch nicht im Pool vorhanden ist, wird es hinzugefügt und die neue Referenz darauf zurückgegeben.

Um dies ein wenig zu veranschaulichen, zeigt das *Listing 1* eine Reihe von Tests, die keiner weiteren Erläuterung bedürfen.

Da „internte" Strings mit dem `==`-Operator verglichen werden können und dieser sicher sehr viel schneller auszuführen ist als die `equals()`-Methode der Klasse `String`, könnte man auf die Idee kommen, dass das Internen aller dynamisch erzeugten Strings zu kürzeren Laufzeiten führt. Scott Oaks *[2]* meint dazu allerdings, dass dies in der Regel nicht der Fall ist, da zwar der Referenz-Vergleich in der Tat schneller ist, dies aber durch die zusätzliche Laufzeit der Aufrufe der `intern()`-Methode ausgeglichen wird.

Bei den Recherchen zu diesem Artikel wurde der Autor in seiner Meinung, dass die Entwickler des OpenJDK ganz unglaublich helle Köpfe sein müssen, sehr bestätigt, da sowohl auf konzeptioneller als auch auf implementierungstechnischer Ebene Unglaubliches, zumindest nach der Werteskala des Autors, geleistet wird. Trotzdem sind auch diese Entwickler nur Menschen und damit fehlbar. Eine recht bekannte, sagen wir semioptimale Entwurfsentscheidung war die ursprüngliche Implementierung der Klasse String. Diese enthielt unter anderem die Instanzvariablen `char[] value`, `int offset` und `int count`, die die einzelnen Zeichen des Strings als Array, den Index des ersten zu verwendenden Zeichens sowie die Gesamtzahl der Zeichen des Strings repräsentierten. Die `String`- und `StringBuffer`-Methode `substring()` war mit dem Ziel der Laufzeit- und Speicherplatzoptimierung derart implementiert, dass der neue Teil-String kein eigenes Zeichen-Array hatte, sondern mithilfe von `value`, `offset` und `count` auf das Ursprungs-Array verwies sowie den Start und die Länge angab. Leider hatte man übersehen, dass diese Lösung verhindert, dass der ursprüngliche String garbage collected werden konnte, wenn er nicht mehr referenziert, wohl aber der Teil-String referenziert wurde. Dieses Memory-Leak führt in unglücklichen Umständen zu Speicherproblemen der JVM. Der Fehler, der die Kennzeichnung „*Memory leak due to String.substring() implementation*" führt, kann in der Fehlerdatenbank des JDK *[3]* nachgelesen werden. Der Fehler wurde mit Java 7 behoben, indem der Teil-String komplett neu angelegt wurde, also inklusive des Zeichen-Arrays. Die Instanzvariablen `offset` und `count` wurden aus der String-Klasse entfernt.

## Compact Strings

Wir haben gerade gesehen, dass die Klasse `String` ein `char`-Array enthält, das die einzelnen Zeichen des Strings repräsentiert. Der JEP 254: *Compact Strings [4]* hatte das Ziel, dies speichereffizienter zu realisieren. Als Java 1995 vorgestellt wurde, war es die erste Sprache, die UTF-16 als Zeichencodierung einsetzte, um dem Anspruch einer weltweit Verwendung findenden Sprache gerecht zu werden. Das JavaDoc der Klasse `Character` enthält einen Abschnitt, in dem die in der jeweiligen Java-Version verwendete Zeichencodierung nachzulesen ist. Mit Java 5 setzte das JDK auf Unicode 4.0, das auch Codierungen enthält, die 32 Bits benötigen. Der 16-Bit-Zeichendatentyp Javas reichte also nicht mehr aus. Den Typ `char` auf 32 Bits zu erweitern, schied aus Kompatibilitätsgründen aus. Die in Unicode definierten Code Points wurden daher auch in Java eingeführt, nachzulesen ebenfalls im JavaDoc der Klasse `Character`. Dies führte beispielsweise dazu, dass das JavaDoc der Methode `String.length()` geändert werden musste, und zwar von Java 5.0 mit

*„Returns the length of this string. The length is equal to the number of 16-bit Unicode characters in the string."*

zu Java 6 mit

*„Returns the length of this string. The length is equal to the number of Unicode code units in the string."*

Java verwendet nun also zwei Bytes pro Zeichen eines Strings, manchmal sogar vier. Oracle untersuchte zu der Zeit Core-Dumps von WebLogic-Instanzen und analysierte die verwendeten Zeichencodierungen. Wenig überraschend kam dabei heraus, dass recht oft Latin-1 als Zeichencodierung ausgereicht hätte, also nur 1 Byte pro Zeichen. Der JEP 254 hat das Ziel, für jeden String eine bedarfsgerechte Speicherform auszuwählen. Enthält der String nur Latin-1-Zeichen, wird pro Zeichen ein Byte verwendet. Wenn nicht zwei oder vier Bytes. Dazu wurde die Instanzvariable `value` zu einem `byte`-Array und es wurde eine weitere Instanzvariable `coder` eingeführt, die die Werte `LATIN1` oder `UTF16` annehmen kann. Diese Umstellung wurde mit Java 9 vollzogen.

Zur Verdeutlichung dieses Sachverhalts dienen ein paar weitere Tests, die im *Listing 2* dargestellt sind. Die Methode `Util.getValueBytes()` ist eine Eigenentwicklung, die über Reflection das Byte-Array zurückgibt.

Auch an dieser Stelle sei noch einmal die Bemerkung erlaubt, dass die Idee und die Umsetzung der geänderten internen String-Implementierung mit Java 9 eine Meisterleistung der Software-Entwicklung waren. Keine API-Änderungen; alle Programme verhielten sich wie vorher, hatten aber ein besseres Laufzeitverhalten und einen geringeren Speicherbedarf. Wir alle haben es nicht bemerkt!

Wie viele Dinge in der JVM ist auch diese speichereffiziente Codierung von Strings konfigurierbar. Falls eine Anwendung zu großen Teilen UTF16-Strings verwendet, ist der Versuch, diese zunächst mit einem Byte zu speichern, um dann doch zu zwei Bytes greifen zu müssen, mit einem unnötigen Mehraufwand verbunden. In diesem Fall kann die JVM-Option `-XX:-CompactStrings` verwendet werden, um generell UTF16-Strings zu verwenden.

Das Kompaktieren von Strings hat eine Vorgeschichte, die einen guten Einblick in die Entwicklungsphilosophie des OpenJDK gibt. Mit Java 6 wurden Strings sowohl als `char[]` als auch als `byte[]` implementiert, also absichtlich redundanter Code erzeugt. Bei der Verwendung der JVM-Option `-XX:+UseCompressedStrings` wurde versucht, Latin-1-Strings in der Byte-Variante zu codieren. Einige String-Methoden waren aber nur für die Char-Version implementiert, sodass die Strings von der einen in die andere Codierung überführt werden mussten. Dieser Aufwand sowie die parallele

```
public class EncodingTest {

  /**
   * Assert that "a" has a one byte representation.
   *
   * See Unicode <a href="http://www.unicode-symbol.com/u/0061.html">latin small letter a</a>
   */
  @Test
  public void oneByteRepresentation() {
     final String str = "\u0061";

     Assertions.assertTrue(str.equals("a"));
     Assertions.assertEquals(1, str.length());
     Assertions.assertEquals(1, str.codePointCount(0, str.length()));
     Assertions.assertEquals(1, Util.getValueBytes(str).length);
  }


  /**
   * Assert that "ś" has a two byte representation.
   *
   * See Unicode <a href="http://www.unicode-symbol.com/u/015B.html">latin small letter s with acute</a>
   */
  @Test
  public void twoBytesRepresentation() {
     final String str = "\u015B";

     Assertions.assertTrue(str.equals("ś"));
     Assertions.assertEquals(1, str.length());
     Assertions.assertEquals(1, str.codePointCount(0, str.length()));
     Assertions.assertEquals(2, Util.getValueBytes(str).length);
  }


  /**
   * Assert that "𩸽" has a four byte representation.
   *
   * See Unicode <a href="http://www.unicode-symbol.com/u/29E3D.html">cjk unified ideograph-29E3D</a>
   */
  @Test
  public void fourBytesRepresentation() {
     final String str = "𩸽"; // hex: 29E3D, int: 171581

     Assertions.assertTrue(str.equals(new String(new int[] { 171581 }, 0, 1)));
     Assertions.assertEquals(2, str.length());
     Assertions.assertEquals(1, str.codePointCount(0, str.length()));
     Assertions.assertEquals(4, Util.getValueBytes(str).length);
  }

}
```

*Listing 2*

Wartung der beiden String-Implementierungen wertete Aleksey Shipilv, ein bekannter JDK-Engineer, mit

*„UseCompressedStrings was really the experimental feature, that was ultimately limited by design, error-prone, and hard to maintain."*

Das Feature Compressed String wurde mit Java 7 wieder aus dem OpenJDK entfernt. Bitte Compact Strings, das seit Java 9 verwendete Speicherschema, und Compressed Strings, ein nur wenige Monate dauernder, nicht geglückter Versuch eines Speicherschemas, nicht durcheinanderbringen.

## String Deduplication

Der JEP 192: String Deduplication in G1 *[5]* hatte das Ziel, die für String-Literale geltende Singularität auch für andere Strings gewährleisten zu können, zumindest nach einem Garbage-Collector-Lauf. Dies wird aber nicht wie bei String-Literalen im String-Pool auf der Ebene der String-Referenz selbst, sondern auf der Ebene der darunter liegenden Array-Referenz realisiert. Die *Abbildung 1a* zeigt zwei Strings, deren jeweilige Zeichen-Arrays identisch sind. Der Garbage Collector erkennt dies, entfernt eines der beiden Arrays und biegt die entsprechende Array-Referenz auf das verbleibende Zeichen-Array um, wie in *Abbildung 1b* dargestellt.

Der Implementierung des JEP 192 ist seit Java 8u20 verfügbar, muss aber mit der Option `-XX:+UseStringDeduplication` explizit angefordert werden. Außerdem ist anzumerken, dass das De-duplizieren nur im Garbage Collector G1 realisiert ist, der allerdings seit längerer Zeit der Default-Collector ist. Der Artikel *„G1: from garbage collector to waste management consultant"* *[6]* nennt für eine durchgeführte Messung eine Verringerung der Heap-Nutzung um 10 %. Da für alle derartigen Alternativen im OpenJDK auch Performanz-Tests durchgeführt werden, die Option aber nicht zu einem Default geworden ist, muss man davon ausgehen, dass der Einsatz nicht generell zu empfehlen ist, sondern bei Bedarf ausprobiert werden sollte. Es gilt wie immer bei Performanzoptimierungen: testen, testen, testen.

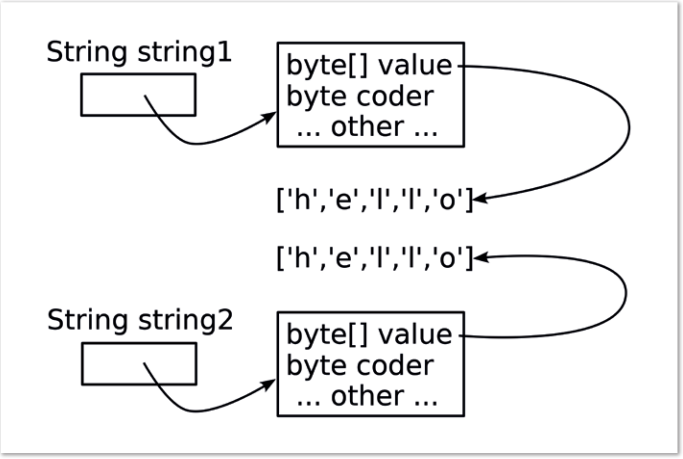



*Abbildung 1a*

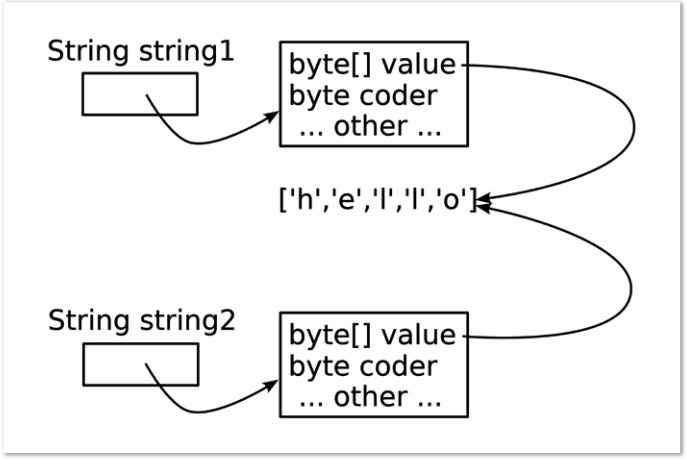



*Abbildung 1b*

## Indify String Concatenation

Der letzte String-relevante JEP, den wir uns anschauen wollen, ist der JEP 280: *Indify String Concatenation [7]*. Die hinter diesem JEP liegende Optimierungsidee hört sich zunächst recht merkwürdig an. Spätere Java-Versionen, also Versionen, die im Augenblick noch nicht existieren, sollen das Konkatenieren von Strings optimiert ausführen können. Hintergrund ist der Umstand, dass die Konkatenierung bereits häufiger optimiert wurde, immer aber auf Compiler-Ebene, sowohl in `javac` als auch im JIT-Compiler, zum Beispiel durch Folgen von `StringBuffer.append()`-Aufrufen. Die Änderungen in den Compilern sind jedoch sehr aufwendig durchzuführen und zudem fehleranfällig. Der JEP beschreibt sie als „*those optimizations, while fruitful, are fragile and difficult to extend and maintain*".

Was wäre, wenn die String-Konkatenierung als möglichst allgemeiner Code formuliert und durch die gerade ausführende JVM, nicht den eventuell vor Jahren verwendeten Compiler, möglichst optimal ausgeführt wird? Genau dies macht der JEP 280. Die allgemeinste Code-Form ist die Verwendung von `invokedynamic`, mit Java 7 durch den JSR 292 eingeführt. Ziel dieses JSR war es, dynamisch getypte Sprachen wie etwa JRuby besser auf der JVM zu unterstützen. Heraus kam allerdings ein allgemeiner Ausführungsmechanismus für Methodenaufrufe. Ebenfalls interessant ist die mit JEP 280 eingeführte Klasse `StringConcatFactory` im Package `java.lang.invoke`. Sie unterstützt ausschließlich diese String-Konkatenation mit `invokedynamic` auf Byte-Code-Ebene, ist also insbesondere nicht dazu gedacht, von uns als Anwendungsentwickler verwendet zu werden. Von außen betrachtet lässt sich dazu nicht mehr sagen. Der Compiler erzeugt für die String-Konkatenation Code, der eventuell in späteren JVMs effizienter als heute ausgeführt werden kann, da er mehr Optimierungspotenzial besitzt.

## Zusammenfassung

Strings sind die am häufigsten verwendeten Java-Objekte. Viele Java-Versionen hatten unter anderem das Ziel, die Verwendung von Strings hinsichtlich der Laufzeit und des Speicherplatzverhaltens zu optimieren. Wir haben den String-Pool vorgestellt, der mithilft, die Singleton-Eigenschaft von Strings zu realisieren. Die JEPs Compact Strings, String Deduplication und Indify String Concatenation wurden ebenfalls erläutern. Auch Versehen, Fehler beziehungsweise Irrwege bei derartigen Optimierungen wurden angesprochen. Der Leser ist nun hoffentlich davon überzeugt, dass die vielen Menschen, die hinter dem OpenJDK stehen, uns immer weiterhelfen, effizientere Java-Software zu schreiben. Wir selbst müssen dazu nichts beitragen, außer weiterhin Java zu nutzen.

## Referenzen


[1] JEP 1: JDK Enhancement-Proposal & Roadmap Process, *https://openjdk.java.net/jeps/1*.

[2] Scott Oaks. Java Performance. O'Reilly 2020, 2nd Edition.

[3] JDK-4637640. *https://bugs.java.com/bugdatabase/view_bug.do?bug_id=4637640*

[4] JEP 254: Compact Strings, *https://openjdk.java.net/jeps/254*.

[5] JEP 192: String Deduplication in G1, *https://openjdk.java.net/jeps/192*.

[6] G1: from garbage collector to waste management consultant, *https://blogs.oracle.com/java/post/g1-from-garbage-collector-to-waste-management-consultant*.

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



**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

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.

Die DOAG Deutsche ORACLE-Anwendergruppe e.V. hält 100 Prozent der Stammeinlage der DOAG Dienstleistungen GmbH. Die DOAG Deutsche ORACLE-Anwendergruppe e.V. wird gesetzlich durch den Vorstand vertreten; Vorsitzender: Björn Bröhl. Die DOAG Deutsche ORACLE-Anwendergruppe 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, Melanie Andrisek, Marcus Fihlon, Markus Karg, Manuel Mauky, Bernd Müller, Benjamin Nothdurft, Daniel van Ross, André Sept

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

Bildnachweis:
Titel: Bild © pch.vector + starline
*https://freepik.com*
S. 10 + 11: Bild © pch.vector
*https://freepik.com*
S. 12: Bild © inkoly
*https://123rf.com*
S. 14 + 15: Bild © pch.vector
*https://freepik.com*
S. 20 + 21: Bild © pch.vector
*https://freepik.com*
S. 22: Bild © Macrovector
*https://stock.adobe.com*
S. 23: Bild © Nuthawut
*https://stock.adobe.com*
S. 26: Bild © rawpixel.com
*https://freepik.com*
S. 32 + 33: Bild © Freepik
*https://freepik.com*
S. 38 + 39: Bild © logturnal
*https://freepik.com*
S. 44 + 45: Bild © denamorado
*https://freepik.com*
S. 50 + 51: Bild © storyset
*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