Java aktuell

iJUG Verbund
www.ijug.eu

Progressive Web Apps
PWAs und ihre Vorteile in der Praxis

Flutter
Von der Entscheidung bis zur Entwicklung

Geteilte Hologramme
Shared Augmented Reality im Überblick

DIE ZUKUNFT IST MOBILE

Ausgabe 03/2020

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

adesso business. people. technology.

Von Anfang an Teil des Java-Teams!

Entwickelt bereits kluge IT-Lösungen bei adesso:
Ihr neuer Kollege Kristof Hierath | Software Engineer

SOFTWARE DEVELOPMENT@adesso

Sie wollen dort einsteigen, wo Zukunft programmiert wird? Dann sind Sie mit einem Start in unserem Java-Team bei adesso genau richtig. Gemeinsam setzen wir herausfordernde Projekte für unsere Kunden um. Dafür brauchen wir Menschen, die Lust haben, ihr Wissen, ihre Talente und ihre Fähigkeiten einzubringen.

Planen und realisieren Sie in interdisziplinären Projektteams anspruchsvolle Anwendungen und Unternehmensportale auf Basis von Java/JavaScript-basierten Technologien als

→ **(Senior) Software Engineer (w/m/d) Java**
→ **Software Architekt (w/m/d) Java**
→ **(Technischer) Projektleiter Softwareentwicklung (w/m/d) Java**

CHANCENGEBER – WAS ADESSO AUSMACHT

Kontinuierlicher Austausch, Teamgeist und ein respektvoller, anerkennender Umgang sorgen für ein Arbeitsklima, das verbindet. So belegen wir nach 2016 auch 2018 den 1. Platz beim Wettbewerb „Deutschlands Beste Arbeitgeber in der ITK“!

Mehr als 650 Software Engineers Java bei adesso, über 120 Schulungen und Weiterbildungen – zum Beispiel in Angular2 oder Spring Boot – sowie ein Laptop und ein Smartphone ab dem ersten Tag warten auf Sie!

GREAT PLACE TO WORK® 2018 Beste Arbeitgeber™ ITK

IHRE BENEFITS – WIR HABEN EINE MENGE ZU BIETEN:

- Welcome Days
- Choose your own Device
- Weiterbildung
- Events: fachlich und mit Spaß
- Sportförderung
- Mitarbeiterprämien
- Auszeitprogramm

Es wird Ihnen bei uns gefallen! Mehr Informationen auf **www.karriere.adesso.de**.
Olivia Slotta aus dem Recruiting-Team freut sich auf Ihre Kontaktaufnahme:
adesso SE | Leona Demiri | T +49 231 7000-7100 | jobs@adesso.de

Unbekannte Kostbarkeiten des SDK Heute: Strings

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.

Strings sind integraler Bestandteil jeder Programmiersprache, da die meisten Daten eines jeden Programms häufig aus Strings bestehen. In Java sind Strings zum einen Instanzen einer ganz normalen Klasse, zum anderen nehmen sie eine besondere Stellung ein, werden explizit in der Sprachbeschreibung genannt und hocheffizient implementiert. Wir wollen uns in dieser Ausgabe unserer unbekannten Kostbarkeiten dieser Besonderheit von Strings widmen, aber auch die API-Erweiterungen der letzten Java-Versionen bekannt machen.

Ein Blick hinter die Kulissen: Strings und String-Literale

Die Klasse `String` ist eine gewöhnliche Klasse, nimmt aber dennoch eine Sonderstellung ein. Da sie sich im Package `java.lang` befindet, gehört sie mit zur Sprache, ist jedoch mit der Klasse `Object` die einzige, der in der Sprachbeschreibung *[1]* ein eigener Abschnitt gewidmet wurde. Im Abschnitt 4.3 werden Referenztypen beschrieben. Die Abschnitte 4.3.2 *The Class Object* und 4.3.3 *The Class String* sind die einzigen Abschnitte, die sich einzelnen Klassen widmen. Dies unterstreicht die zentrale Stellung von Strings.

Warum sind Strings etwas Besonderes? Scott Oaks, anerkannter Java-Fachmann und Autor mehrerer Bücher mit Java-Bezug, schreibt in seinem sehr empfehlenswerten Buch *Java Performance [2]*: „Strings are, far and away, the most common Java object; your application's heap is almost certainly filled with them." Kein Wunder also, dass Strings auf Bibliotheks-, Compiler-, JIT- und JVM-Ebene eine besondere Aufmerksamkeit genießen. Wenn Strings und String-Operationen effizient implementiert sind, kommt das der Laufzeit von Java-Anwendungen insgesamt zugute.

Beginnen wir mit unseren Betrachtungen ganz am Anfang. In der Version 1.0 von Java wurden Strings intern als Character-Array (`char[]`) implementiert. Um den Speicherbedarf zu minimieren, sind String-Literale in der virtuellen Maschine Unikate. Die Sprachbeschreibung *[1]* sagt in Abschnitt 3.10.5 *String Literals* dazu: *„String literals – or, more generally, strings that are the values of constant expressions – are 'interned' so as to share unique instances, using the method String.intern"*. Dazu gibt es einen String-Pool, der diese Unikate enthält. Wird eine neue Klasse in die VM geladen, so wird für jedes String-Literal im Kompilat dieser Klasse (.class-Datei) geprüft, ob es sich schon im String-Pool befindet. Falls ja, wird die Referenz auf diesen Pool-Eintrag verwendet, falls nein, wird das Literal eingetragen und diese neue Referenz verwendet. Der String-Pool ist in C innerhalb der VM als Hash-Map fester Größe implementiert. Die Größe kann beim VM-Start mit `--XX:StringTableSize=N` festgelegt werden. Die Default-Größe betrug bis Java 7u40 1009, danach 60013 Buckets. Bis Java 6 befand sich der String-Pool im PermGen. Da mit Java 8 der PermGen durch den Metaspace ersetzt werden sollte, wurde bereits mit Java 7 der String-Pool vom PermGen in den Heap verlagert. Der String-Pool wird *garbage-collected*. Da die String-Literale aber Referenzen innerhalb von Methoden darstellen, können nur String-Literale gelöscht werden, deren Methoden respektive Klassen entladen wurden. Pool-Einträge, die durch die noch vorzustellende `intern()`-Methode in den Pool aufgenommen wurden, können ganz normal *garbage-collected* werden.

Das explizite Internen von Strings

String-Literale sind Unikate im String-Pool, wie wir bereits gesehen haben. Andere Strings können programmatisch in den String-Pool aufgenommen und somit zu Unikaten gemacht werden. Die entsprechende Methode der Klasse `String` ist `intern()`. Diese prüft, ob der String bereits im Pool vorhanden ist. Falls ja, wird diese Referenz zurückgegeben, falls nein, wird der String eingetragen und diese neue Referenz zurückgegeben. Um dies zu verdeutlichen, werden in *Listing 1* die möglichen Fälle durchgespielt.

Der String „`Hello, World!`“ kommt insgesamt fünf Mal in *Listing 1* vor. Drei Mal bei der Initialisierung der Variablen, zwei Mal im Test 2. Zur Laufzeit in der VM existiert dieser String im String-Pool genau ein Mal und alle fünf Literalverwendungen verweisen auf diesen einen Pool-Eintrag. Der String existiert noch ein weiteres Mal auf dem Heap, da der String-Konstruktor bei der Initialisierung von `str3` ein Heap-Objekt als Kopie des Konstruktorparameters erzeugt. Die Tests 1 und 2 zeigen, dass alle Literalverwendungen sowie die Variablen `str1` und `str2` auf den Pool-Eintrag verweisen. Die Tests 3 und 4 belegen den oben wiedergegebenen Teil der Definition eines String-Literals mit „values of constant expressions“. Die bereits zur Compile-Zeit verbundenen Literale entsprechen dem diesbezüglichen Literal. Dass dies für andere Ausdrücke, zum Beispiel Ausdrücke mit Variablen, nicht zutrifft, zeigen die Tests 5 und 6. Die aneinandergehängten Variablen sind zwar „Equals“, aber nicht identisch. Dass solche Heap-Strings mit der `intern()`-Methode in den String-Pool überführt werden können, zeigt der Test 7. Die Tests 8 und 9 belegen schließlich die obige Aussage, dass sich `str3` im Heap und nicht im String-Pool befindet.

```
String str1 = "Hello, World!";
String str2 = "Hello, World!";
String str3 = new String("Hello, World!");
/* 1 */ assertSame(str1, str2);
/* 2 */ assertSame("Hello, World!", "Hello, World!");
/* 3 */ assertSame("Hel" + "lo", "Hel" + "lo");
/* 4 */ assertSame("Hel" + "lo", "Hello");
/* 5 */ assertEquals(str1 + str1, str2 + str2);
/* 6 */ assertNotSame(str1 + str1, str2 + str2);
/* 7 */ assertSame((str1 + str1).intern(),
                   (str2 + str2).intern());
/* 8 */ assertNotSame(str1, str3);
/* 9 */ assertEquals(str1, str3);
```

Listing 1

Schnittstelle ohne Implementierungsbezug

Dass das JDK von erfahrenen Entwicklern definiert wurde, weiterentwickelt wird und diese die Grundsätze des Software-Engineerings beherzigen, kann man an Strings recht gut belegen. Es gab zumindest zwei fundamentale Änderungen in der String-Implementierung, die praktisch keine Auswirkungen auf die Schnittstelle der Klasse `String` hatten. In der Version 1.0 sind Strings intern als Character Array (`char[]`) definiert. Ein char ist ein vorzeichenloser, ganzzahliger Datentyp mit Werten zwischen 0 und 65535 (\uffff), die Unicode-Zeichen repräsentieren und einen Speicherbedarf von 16 Bits haben. Mit Java 5 wurde Unicode 4.0 eingeführt, womit Zeichen-Codierungen größer als \uffff zu unterstützen waren. Diese sogenannten „supplementary“ Character wurde ohne eine Änderung des `char`-Datentyps und die damit unweigerlich verbundenen Inkompatibilitäten eingeführt, indem sie als ein Paar von Code-Point-Werten, sogenannte *Surrogate*, repräsentiert werden. Seit Java 5 kann also ein Unicode-Zeichen zwei `char`-Werte zur Darstellung benötigen. An uns als Entwickler mit typischerweise europäischen Zeichensätzen im Tagesgeschäft gingen die zusätzlichen Methoden in der Klasse String unbemerkt vorbei. In der Sprachbeschreibung *[3]* wurden lediglich die letzten Wörter bei der Definition von Strings von „Instances of class String represent sequences of Unicode characters“ in „... Unicode code points“ geändert.

Eine wesentlich tiefgreifende Änderung wurde mit Java 9 realisiert. Die String-Implementierung wurde intern auf ein Byte Array (`byte[]`) umgestellt. Wie wir gerade gesehen haben, wurden Zeichen in 16 Bits codiert. Einige chinesische und japanische Zeichen benötigen sogar 32 Bits. Für sehr viele Anwendungen würden jedoch auch 8 Bits ausreichen. Für ein einzelnes Zeichen ist das unerheblich, für Strings nicht. Der JEP 254: *Compact Strings [4]* hatte das Ziel, den Speicherbedarf der String-Implementierung zu verringern. Seit Java 9 werden Strings intern in einem Byte Array statt in einem Character Array (ein Field mit Name `value`) gehalten und das Flag `coder` identifiziert die Codierung des Byte Array. Das Flag kann die beiden Werte LATIN1 und UTF16 annehmen. Besteht ein String (Literal oder eingelesen) aus Zeichen, die eine 16- oder gar 32-Bit-Codierung benötigen, so wird UTF16 verwendet, falls nicht LATIN1.

String-Deduplication

Das Ziel des JEP 192: *String Deduplication [5]* ist: „Reduce the Java heap live-data set by enhancing the G1 garbage collector so that duplicate instances of String are automatically and continuously deduplicated.“ Unter einem Duplikat sind zwei gleiche Strings (Methode `equals()`) zu verstehen. Die Deduplizierung findet jedoch nicht auf den String-Objekten, sondern auf den dahinter existierenden Arrays statt. Da JEP-Nummern in der Regel einfach hochgezählt werden, begannen die Arbeiten zum JEP 192 offensichtlich zu einem Zeitpunkt, als Stings noch als Character-Arrays implementiert waren; dies ist so auch im JEP nachlesbar. Mittlerweile, nach Umsetzung des JEP 254, werden Byte Arrays dedupliziert. Konzeptionell wird einfach eine der beiden Array-Referenzen auf das andere Array umgesetzt und das nicht mehr referenzierte Array kann speicherbereinigt werden. Auch dieses Verhalten kann man sehr anschaulich überprüfen, wie der Code in *Listing 2* zeigt

```
int random = new Random().nextInt();
String string1 = new String("some string " + random);
String string2 = new String("some string " + random);
Field field = String.class.getDeclaredField("value");
field.setAccessible(true);

Object object1 = field.get(string1);
Object object2 = field.get(string2);
/* 1 */ assertNotSame(object1, object2);

System.gc();
Thread.sleep(1000);

Object object3 = field.get(string1);
Object object4 = field.get(string2);
/* 2 */assertSame(object3, object4);
```

Listing 2

```
// Java 8

public static String join(CharSequence delimiter, CharSequence... elements)
// Returns a new String composed of copies of the CharSequence elements joined together with a copy of the
specified delimiter.

public static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)
// Returns a new String composed of copies of the CharSequence elements joined together with a copy of the
specified delimiter.


// Java 9

public IntStream chars()
// Returns a stream of int zero-extending the char values from this sequence. Any char which maps to a surrogate
code point is passed through uninterpreted.

public IntStream codePoints()
// Returns a stream of code point values from this sequence. Any surrogate pairs encountered in the sequence are
combined as if by Character.toCodePoint and the result is passed to the stream.


// Java 11

public String strip()
// Returns a string whose value is this string, with all leading and trailing white space removed.

public String stripLeading()
// Returns a string whose value is this string, with all leading white space removed.

public String stripTrailing()
// Returns a string whose value is this string, with all trailing white space removed.

public boolean isBlank()
// Returns true if the string is empty or contains only white space codepoints, otherwise false.

public Stream<String> lines()
// Returns a stream of lines extracted from this string, separated by line terminators.

public String repeat(int count)
// Returns a string whose value is the concatenation of this string repeated count times.


// Java 12

public String indent(int n)
// Adjusts the indentation of each line of this string based on the value of n, and normalizes line termination
characters.

public <R> R transform(Function<? super String,? extends R> f)
// This method allows the application of a function to this string. The function should expect a single String
argument and produce an R result.

public Optional<String> describeConstable()
// Returns an Optional containing the nominal descriptor for this instance, which is the instance itself.

public String resolveConstantDesc(MethodHandles.Lookup lookup)
// Resolves this instance as a ConstantDesc, the result of which is the instance itself.


// Java 13

@Deprecated(forRemoval=true, since="13")
public String stripIndent()
// Returns a string whose value is this string, with incidental white space removed from the beginning and end of
every line.

@Deprecated(forRemoval=true, since="13")
public String translateEscapes()
// Returns a string whose value is this string, with escape sequences translated as if in a string literal.

@Deprecated(forRemoval=true, since="13")
public String formatted(Object... args)
// Formats using this string as the format string, and the supplied arguments.
```

Listing 3

Zunächst erzeugen wir mithilfe eines Zufallszahlengenerators zwei Strings, deren implementierende Arrays dieselben Inhalte enthalten, die aber verschieden in der Semantik von `assertNotSame()` sind, wie Test 1 zeigt. Danach rufen wir den Garbage Collector auf und müssen ihm, da er in einem anderen Thread läuft, auch genug Zeit geben, sein Werk – das Deduplizieren – zu beenden. Danach existiert nur noch eines der beiden Arrays und beide `value`-Fields verweisen darauf, wie Test 2 beweist. Falls Sie den Code selbst ausführen wollen, achten Sie darauf, dass G1 als Garbage Collector verwendet wird (`-XX:+UseG1GC`, Default seit Java 9) und String-Deduplication aktiviert ist (`-XX:+UseStringDeduplication`).

Die Idee unserer Kolumne ist, Ihnen unbekannte Kostbarkeiten des SDK vorzustellen, um zusätzliche Frameworks vermeiden und Aufwände minimieren zu können. Mit den bisherigen Ausführungen zu Strings verhält es sich anders. Sie müssen nichts tun, trotzdem verringert sich die Laufzeit und der Speicherbedarf Ihrer Anwendung reduziert sich, wenn Sie zu einer neueren Version migrieren, die die vorgestellten Optimierungen eingebaut hat. Es ist, wie wir meinen, eine beruhigende Vorstellung, dass dies von Zeit zu Zeit ganz ohne unser Zutun passiert. An dieser Stelle vielen Dank an alle Beteiligten, die Java voranbringen.

Neue Methoden der Klasse String

Um nicht nur automatisch stattfindende Verbesserungen zu beschreiben, wollen wir unseren Artikel mit API-Änderungen beschließen, die auch für die Klasse `String` regelmäßig stattfinden. In *Listing 3* sind diese beginnend mit Java 8 aufgeführt. Neben den Signaturen zeigt *Listing 3* auch die jeweils erste Zeile des JavaDoc der Methode. Da diese Beschreibungen sehr anschaulich sind, verzichten wir auf Beispiele. Ausnahmen sind lediglich die in Java 12 hinzugefügten Methoden `describeConstable()` und `resolveConstantDesc()`. Sie erlauben die Verwendung des Constant Pool der JVM und sind für die meisten Leser wahrscheinlich nicht interessant, wenn es um die Realisierung fachlicher Anforderungen geht. Wir verzichten auf ein Beispiel. Eine Besonderheit nehmen die mit Java 13 eingeführten Methoden ein, die zugleich als `@Deprecated` annotiert wurden. Dies hängt mit dem Preview Mode für Text-Blocks zusammen. Trotzdem können die Methoden verwendet werden, ohne beim Kompilieren oder Ausführen die Option `--enable-preview` verwenden zu müssen.

Zusammenfassung

Die Klasse `String` ist eine zentrale Klasse des SDK. Durch die häufige Verwendung von Strings in praktisch jeder Anwendung ist eine möglichst optimale Implementierung eine Grundvoraussetzung für hohe Performanz und geringe Speicheranforderungen. Die Ingenieure des SDK haben daher sowohl auf Sprach-, als auch auf Implementierungsebene in den letzten Jahren erhebliche Optimierungen vorgenommen, die wir vorgestellt haben. Sie gehören unzweifelhaft zu den unbekannten Kostbarkeiten des SDK und in das Wissen eines jeden Java-Entwicklers. Abschließend haben wir noch eine Reihe von Methoden der Klasse `String` aufgeführt, die seit Java 8 in die Klasse aufgenommen wurden *(siehe Listing 3)*.

Referenzen

[1] James Gosling, Bill Joy, Guy Steele. The Java Language Specification, Addision-Wesley, 1996.

[2] Scott Oaks. Java Performance – The Definitive Guide, O´Reilly, 2014.

[3] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith. The Java Language Specification, Java SE 13 Edition. 2019.

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

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

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.

Java aktuell

Mehr Informationen zum Magazin und Abo unter:

https://www.ijug.eu/de/java-aktuell

iJUG Verbund
www.ijug.eu

Eventpartner: AOUG SOUG swiss oracle user group iJUG Verbund ORACLE