6 Neues und Änderungen in Java 10

Nur rund sechs Monate nach Java 9 als umfangreiches und bedeutsames Java-Update erblickte Java 10 im März 2018 das Licht der Welt. Aufgrund der Kürze der Zeit zwischen den beiden Releases enthält Java 10 lediglich wenige Änderungen und Erweiterungen. Die am meisten beachtete Neuerung ist vermutlich die sogenannte Local Variable Type Inference, besser bekannt als das reservierte Wort[1] var. Zudem wurden einige APIs ergänzt, vor allem bei Collections, Kollektoren und der Klasse java.util.Optional<T>. Darüber hinaus finden sich noch Neuerungen im Versionsschema sowie diverse Kleinigkeiten in unterschiedlichen APIs. Alle diese Änderungen werden nachfolgend in jeweils eigenen Abschnitten anhand von Beispielen eingeführt und besprochen.

Bitte beachten Sie, dass es sich bei Java 10 ebenso wie bei Java 9 lediglich um ein kurzzeitig verfügbares Release handelt, das mit dem Erscheinen von Java 11 im September 2018 nicht länger Support und Bugfixes erhält.

6.1 Syntaxerweiterung var

Wie einleitend erwähnt, bietet Java 10 die Local Variable Type Inference als Syntaxerweiterung. Diese erlaubt es, mithilfe von var auf die explizite Typangabe auf der linken Seite einer Variablendefinition zu verzichten. Um dort var als Typplatzhalter nutzen zu können, muss sich der konkrete Typ für eine lokale Variable anhand der Definition auf der rechten Seite der Zuweisung vom Compiler ermitteln lassen.

Einführende Beispiele

Schauen wir uns einige einführende Beispiele für die Kurzschreibweise mit var für Variablendefinitionen an:

```
var name = "Peter";                // var => String
var chars = name.toCharArray();    // var => char[]

var mike = new Person("Mike", 47); // var => Person
var hash = mike.hashCode();        // var => int
```

[1] Es ist tatsächlich kein Schlüsselwort, sondern wird nur speziell ausgewertet.

Im Zusammenhang mit generischen Containern spielt die Local Variable Type Inference ihre Vorteile aus:

```
// var => ArrayList<String>
var names = new ArrayList<String>();
names.add("Tim");
names.add("Tom");
names.add("Jerry");

// var => Map<String, Long>
var personAgeMapping = Map.of("Tim", 47L, "Tom", 12L,
                              "Michael", 47L, "Max", 25L);
```

Insbesondere wenn die Typangaben mehrere generische Parameter umfassen, kann `var` den Sourcecode deutlich kürzer und mitunter lesbarer machen. Betrachten wir als Beispiel eine Verschachtelung von Typen analog zu den folgenden:

- `Set<Map.Entry<String, Long>>`
- `Map<Character, Set<Map.Entry<String, Long>>>`

In solchen Fällen spart `var` einiges an Schreibarbeit – zusätzlich erfolgt hier noch ein statischer Import verschiedener Kollektoren, um die Lesbarkeit zu steigern:

```
// var => Set<Map.Entry<String, Long»
var entries = personAgeMapping.entrySet();

// var => Map<Character, Set<Map.Entry<String, Long>>>
var filteredPersons = personAgeMapping.entrySet().stream().
                                   collect(groupingBy(firstChar,
                                           filtering(isAdult, toSet())));
```

Im Beispiel werden zur Beschreibung der Gruppierung und zur Filterung folgende zwei Lambdas genutzt, worauf ich gleich nochmals genauer eingehe:

```
Function<Map.Entry<String, Long>, Character> firstChar =
                                             entry -> entry.getKey().charAt(0);

Predicate<Map.Entry<String, Long>> isAdult = entry -> entry.getValue() >= 18;
```

Hilfestellung in IDEs

So angenehm die Kurzschreibweise in der Regel auch ist, so wünschenswert ist manchmal eine Expansion in oder ein Hinweis auf den konkreten Typ. In beiden Fällen ist der intelligente Tooltip in Eclipse hilfreich, wie es folgende Abbildung 6-1 zeigt.

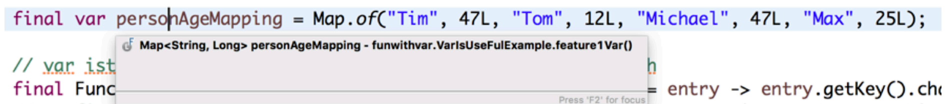

Abbildung 6-1 *Hilfestellung zu* `var` *in Eclipse*

Vermutlich eher selten besteht das Bedürfnis, doch wieder den konkreten Typ statt `var` zu nutzen. Praktischerweise existieren Quick Fixes in den gebräuchlichen IDEs, um zwischen konkretem Typ und `var` leicht hin- und herzuwechseln.

Lambda-Ausdrücke und `var`

Eine Kleinigkeit sollten wir noch betrachten: Im vorherigen Beispiel kommen beim Aufbereiten der Map folgende zwei Lambdas zum Einsatz, um die Funktionalität zu realisieren:

```
Function<Map.Entry<String, Long>, Character> firstChar =
                                             entry -> entry.getKey().charAt(0);

Predicate<Map.Entry<String, Long>> isAdult = entry -> entry.getValue() >= 18;
```

Wäre es nicht wünschenswert, auch hier die Typangabe mit `var` abzukürzen? Eigentlich ja! Warum dies nicht geht, erkläre ich im Folgenden.

Der Compiler kann rein auf Basis dieses Lambdas den konkreten Typ nicht ermitteln. Somit ist keine Umwandlung in `var` möglich, sondern dies führt vielmehr zur Fehlermeldung »`lambda expression needs an explicit target-type`«. Wollte man `var` trotzdem nutzen, so müsste man folgenden Cast einfügen:

```
var isAdultVar =
    (Predicate<Map.Entry<String, Long>>) entry -> entry.getValue() >= 18;
```

Insgesamt sieht man, dass `var` für Lambda-Ausdrücke eher ungeeignet ist. Das ist insofern schade, weil hier einige Schreibarbeit gespart werden könnte.

Beschränkungen

Rekapitulieren wir kurz: `var` ist für lokale Variablen gedacht, die direkt initialisiert werden. Damit ist es im Speziellen auch für Variablen in `for`-Schleifen und try-with-resources geeignet. Darüber hinaus scheint mitunter wünschenswert, `var` zur Deklaration von Attributen, Parametern oder Rückgabetypen nutzen zu können. Das ist ebenso wie für Lambdas jedoch nicht möglich, weil der Typ vom Compiler nicht eindeutig ermittelt werden kann.

Restriktion auf exakten Typ Beim Einsatz von `var` sollte man wissen, dass immer der exakte Typ verwendet wird und nicht ein Basistyp, wie man es für Collections getreu dem Paradigma »program against interfaces« sehr gerne macht. Beachten Sie bitte, dass im Folgenden die Variable `names` deshalb nicht vom Typ `List<String>` ist, sondern vom Typ `ArrayList<String>`:

```
// var => ArrayList<String>
var names = new ArrayList<String>();
names = new LinkedList<String>(); // Compile Error
```

Versucht man, eine Instanz einer `LinkedList<String>` an die Variable `names` zuzuweisen, so erzeugt dies die Fehlermeldung: »`type mismatch: cannot convert from LinkedList<String> to ArrayList<String>`«. Der Grund sollte Ihnen mittlerweile bekannt sein: Es kommt zum Fehler, weil der durch `var` repräsentierte Typ `ArrayList<String>` und eben nicht `List<String>` ist, wie man es vielleicht erwarten würde.

Weitere syntaktische Besonderheiten Es gibt weitere Dinge, die zu Kompilierfehlern führen. Das sind eine fehlende Wertangabe und auch eine fehlende Typangabe bei direkten Array-Initialisierungen:

```
var justDeclaration;              // keine Wertangabe / Definition
var numbers = {0, 1, 2};          // fehlende Typangabe
```

Fallstrick

Auf einen Fallstrick beim Einsatz von `var` möchte ich unbedingt noch eingehen: Manchmal ist man versucht, ohne viel nachzudenken die Typangabe auf der linken Seite direkt mit `var` zu ersetzen. Schauen wir auf ein harmlos wirkendes Beispiel einer auf String typisierten Liste:

```
List<String> names = new ArrayList<>();
names.add("Expected");
// names.add(42); // Compile error
```

Hier nutzen wir auf der rechten Seite den sogenannten Diamond Operator, der es erlaubt, auf die explizite Angabe des generischen Typs zu verzichten. Das ist möglich, weil dieser aus der linken Seite der Variablendeklaration, genauer der Typangabe, hergeleitet werden kann. Nehmen wir an, wir würden nun die Angabe von `List<String>` durch `var` ersetzen und den zweiten Aufruf von `add()` einkommentieren:

```
var mixedContent = new ArrayList<>();
mixedContent.add("Strange with var");
mixedContent.add(42);
```

Kompiliert und funktioniert das? Und wenn ja, was ist daran problematisch? Tatsächlich produziert das Ganze keinen Kompilierfehler. Wie kommt das? Aufgrund des Diamond Operators bzw. der nicht vorhandenen Typangabe stehen dem Compiler nicht ausreichend Typinformationen zur Verfügung: Deswegen wird `java.lang.Object` als generischer Parameter genutzt und aus der zuvor auf String typisierten Liste wird – wohl eher unerwartet – eine `ArrayList<Object>`!

Die gezeigte Modifikation stellt einen Flüchtigkeitsfehler dar, der schwerwiegende Folgen haben kann. Genau deshalb steht der Umwandlungs-Quick-Fix in den IDEs nur dann zur Verfügung, wenn auf der linken Seite der exakte Typ angegeben wurde. In der Praxis ist dies aber durch die oftmals sinnvolle Designregel, gegen Interfaces zu programmieren, vor allem für Collections eher selten der Fall: Regelkonform arbeitet

man bei der Variablendeklaration vielfach mit einem Interface statt mit einer konkreten Klasse. Bevor Sie etwas übereifrig dann selbst Hand anlegen und `var` nutzen, fügen Sie bitte die Typangabe auf der rechten Seite ein, um Probleme durch fehlende Typsicherheit direkt zu umgehen.

6.2 API-Neuerungen

Nachfolgend gehe ich zunächst auf das Erzeugen unveränderlicher Kopien von Listen, Sets und Maps ein. Danach zeige ich Erweiterungen im Stream-API: Spezielle Kollektoren können nun unveränderliche Ergebnisdatenstrukturen liefern. Zudem behandle ich eine kleine, aber feine Erweiterung in der Klasse `java.util.Optional<T>`. Im Anschluss gehe ich auf Wissenswertes zu den Änderungen an der Versionierung ein. Abschließend werden verschiedene kleinere API-Neuerungen vorgestellt.

6.2.1 Unveränderliche Kopien von Collections

Bevor wir auf die Neuerung zum Erzeugen unveränderlicher Kopien von Collections in Java 10 schauen, wollen wir zunächst einen Blick zurück wagen. Mit Java 9 wurden sogenannte Collection Factory Methods eingeführt, die es erlauben, unveränderliche Collections zu erzeugen. Dazu dienen verschiedene statische `of()`-Methoden sowie für Maps zusätzlich eine `ofEntries()`-Methode.

Einführendes Beispiel

Nachfolgend ist ein Beispiel zum Erzeugen eines unveränderlichen Sets dargestellt:

```
private Set<String> createImmutableSetJdk9Style()
{
    return Set.of("Tim", "Tom", "Jerry");
}
```

Alternativ kann man mit `asList()` basierend auf einer festgelegten Wertefolge eine korrespondierende unveränderliche Liste erzeugen – jedoch existiert leider nichts Ähnliches für Sets und auch nicht für Maps.

```
final List<String> names = Arrays.asList("Tim", "Tom", "Jerry");
```

In keinem Fall war es möglich, von einem bereits existierenden Container eine unveränderliche Kopie zu erzeugen. Als Abhilfe wurde oftmals folgendes Idiom eines Konstruktoraufrufs genutzt:

```
final List<String> newCopyOfCollection = new ArrayList<>(names);
```

Basierend auf den Eigenschaften einer `ArrayList<E>` ist die so entstehende Kopie allerdings veränderlich – selbst dann, wenn die ursprüngliche Datenstruktur unveränderlich war.

Utility-Funktionalität Unmodifiable Wrapper

Auch der bis einschließlich Java 8 verfügbare »Umweg« über die Utility-Funktionalität in Form der überladenen Methoden `unmodifiableXyz()` (`Xyz` = `List`, `Set`, `Map`) führt nicht zu einer unveränderlichen Kopie: Es entsteht dadurch lediglich ein Wrapper um die ursprüngliche Datenstruktur. Zudem ist recht viel Schreibaufwand nötig, wie es exemplarisch für ein `HashSet<E>` mit drei Elementen gezeigt ist:

```
private Set<String> createImmutableSetOldStyle()
{
    final Set<String> names = new HashSet<>();
    names.add("Tim");
    names.add("Tom");
    names.add("Jerry");

    return Collections.unmodifiableSet(names);
}
```

Erweiterung in Java 10

Für den Fall, dass man eine unveränderliche Kopie einer Liste benötigt, macht die in Java 10 neu eingeführte Methode `copyOf()` das Leben deutlich einfacher als noch mit Java 9. Das zeigt folgender Aufruf eindrucksvoll:

```
final List<String> newImmutableCopy = List.copyOf(names);
```

Selbstverständlich gibt es Analogien für Sets und Maps. Das sind die Methoden `Set.copyOf(originalSet)` und `Map.copyOf(originalMap)`.

Beispiel

Betrachten wir die neue Kopierfunktionalität für die jeweiligen Datenstrukturen:

```
var names = List.of("Tim", "Tom", "Jerry");
List<String> copyOfNames = List.copyOf(names);
System.out.println("copyOfList: " + copyOfNames.getClass());

var colors = Set.of("Red", "Green", "Blue");
Set<String> copyOfColors = Set.copyOf(colors);
System.out.println("copyOfSet:  " + copyOfColors.getClass());

var personAgeMapping = Map.of("Tim", 47L, "Tom", 12L, "Mike", 47L, "Max", 25L);
Map<String, Long> copyOfMap = Map.copyOf(personAgeMapping);
System.out.println("copyOfMap:  " + copyOfMap.getClass());
```

Durch die obigen Anweisungen kommt es zu folgenden Ausgaben, die zeigen, dass hier die mit Java 9 eingeführten Immutable-Collection-Klassen verwendet werden, wie dies auch beim Aufruf der Collection Factory Methods der Fall ist.

```
copyOfList: class java.util.ImmutableCollections$ListN
copyOfSet:  class java.util.ImmutableCollections$SetN
copyOfMap:  class java.util.ImmutableCollections$MapN
```

Wir erhalten hier also tatsächlich unveränderliche Kopien. Wird trotzdem eine Modifikation versucht, so wird diese verhindert und eine `UnsupportedOperationException` ist die Folge.

6.2.2 Immutable Collections aus Streams erzeugen

Das Stream-API war schon bei seiner Einführung mit Java 8 recht umfangreich und wurde mit Java 9 erweitert. Allerdings ließen sich bislang keine unveränderlichen Datenstrukturen erzeugen. Dies wird seit Java 10 durch die Methoden `toUnmodifiableList()`, `toUnmodifiableSet()` und `toUnmodifiableMap()` der Klasse `java.util.stream.Collectors` möglich. Einführend schauen wir auf ein Beispiel für unveränderliche Listen und Sets.

Unveränderliche Listen und Sets

Basierend auf einer Liste mit teils doppelten Namen wollen wir eine unveränderliche Kopie als Liste bzw. als Set erzeugen. Die oben genannten Methoden haben wir der besseren Lesbarkeit halber statisch importiert:

```
var names = List.of("Tim", "Tom", "Mike", "Peter", "Tom", "Tim");

var immutableNames = names.stream().collect(toUnmodifiableList());
System.out.println("immutableNames type: " + immutableNames.getClass());

var uniqueImmutableNames = names.stream().collect(toUnmodifiableSet());
System.out.println("uniqueImmutableNames content: " +
                   uniqueImmutableNames);
System.out.println("uniqueImmutableNames type: " +
                   uniqueImmutableNames.getClass());
```

Diese Zeilen geben Folgendes aus:

```
immutableNames type: class java.util.ImmutableCollections$ListN
uniqueImmutableNames content: [Peter, Mike, Tim, Tom]
uniqueImmutableNames type: class java.util.ImmutableCollections$SetN
```

Beachten Sie bitte, dass die Reihenfolge der Elemente bei Ihnen durchaus abweichen kann – diese ist durch `Set.of()` nicht festgelegt.

Hinweis Interessanterweise verhält sich hier der Kollektor `toUnmodifiableSet()` anders als die Methode `of()` aus dem Interface `Set<E>`: Diese würde nämlich bei Duplikaten eine Exception auslösen. Das geschieht hier für die Duplikation von »Tim« und »Tom« nicht.

Unveränderliche Maps

Kommen wir schließlich zur Erzeugung unveränderlicher Maps. Als Beispiel hierzu bereiten wir folgendermaßen aus den vorherigen Daten ein Histogramm auf, das die Häufigkeit der jeweiligen Namen repräsentiert.

```
Function<String, Long> valueMapper = value -> 1L;
BinaryOperator<Long> mergeFunction = (count, inc) -> count + inc;

var nameCountMap = names.stream().collect(toUnmodifiableMap(Function.identity(),
                                                            valueMapper,
                                                            mergeFunction));

System.out.println("content: " + nameCountMap);
System.out.println("type:    " + nameCountMap.getClass());
```

Damit erhalten wir folgende Ausgaben:

```
content: {Peter=1, Mike=1, Tim=2, Tom=2}
type:    class java.util.ImmutableCollections$MapN
```

Erneut sei darauf hingewiesen, dass die Reihenfolge der Elemente bei Ihnen abweichen kann, weil diese durch `Map.of()` nicht festgelegt ist.

Aber interessanter ist die Frage: Wie funktioniert die Magie? Die zuvor gezeigten Lambdas machen eine Identitätsabbildung für die Schlüssel, setzen den Startwert auf 1 und vereinigen dann die Werte über eine sogenannte Merge-Funktion. Diese definiert, wie Kollisionen bei gleichem Key aufgelöst werden.

6.2.3 Erweiterung in der Klasse `Optional`

Die Klasse `java.util.Optional<T>` war eine enorme Bereicherung in Java 8: Lange hatte sich die Java-Gemeinde gewünscht, optionale Werte, vor allem für Rückgaben[2], geeignet modellieren zu können. Mit Java 9 wurde das API durch die drei Methoden `ifPresentOrElse()`, `or()` und `stream()` vervollständigt. Damit schien das API recht komplett. Allerdings hat man sich bei Oracle an einer Unschönheit gestört: an der Methode `get()`. Wieso?

Die Antwort ist einfach: Die Methode `get()` zum Zugriff auf den Wert eines `Optional<T>` sieht zu harmlos aus. Vermutlich dadurch findet man in der Praxis mitunter einen Aufruf von `get()` ohne vorherige Prüfung auf Existenz eines Werts mit `isPresent()`. Das führt dann aber bei einem nicht vorhandenen Wert zu einer `java.util.NoSuchElementException`. Normalerweise erwartet man beim Aufruf einer `get()`-Methode allerdings nicht unbedingt, dass diese eine Exception auslöst. Um diesen Sachverhalt im API direkt ausdrücken zu können, wurde die Methode `orElseThrow()` als Alternative zu `get()` in das JDK aufgenommen, die bei Vorhandensein, den Wert liefert und ansonsten eben eine `NoSuchElementException` auslöst.[3]

[2] Ob es auch zur Definition von Attributen genutzt werden sollte oder nicht, ist in der Entwicklergemeinde umstritten. Laut Oracle soll es nur für Rückgaben zum Einsatz kommen.

[3] Das gilt ebenso für die Spezialisierungen von `Optional<T>` für die primitiven Typen `double`, `int` und `long`.

Hinweis: Anmerkung zur Sinnhaftigkeit

Ob die neue Methode wirklich benötigt wird, kann man infrage stellen. Zum einen ist es auch beim `java.util.Iterator<T>` so, dass ein Aufruf von `next()` ohne vorherigen Aufruf von `hasNext()` eine Exception auslöst. Zum anderen hätte der Name besser `getOrElseThrow()` lauten sollen, um Verwechslungen mit `orElse()` oder `orElseGet()` zu vermeiden.

Beispiel

Nachfolgend lernen wir in der JShell die Neuerung in `Optional<T>` kennen:

```
jshell> Optional<String> optValue = Optional.of("ABC");
optValue ==> Optional[ABC]

jshell> String value = optValue.orElseThrow();
value ==> "ABC"

jshell> Optional<String> empty = Optional.empty();
empty ==> Optional.empty

jshell> empty.orElseThrow();
|  java.util.NoSuchElementException thrown: No value present
```

Hinweis: Kleinere Experimente mit der JShell

Auch wenn man über den Nutzen der JShell unterschiedlicher Meinung sein kann, so ist jedoch ein unbestreitbarer Vorteil davon, dass sich kleine Programmschnipsel direkt ausprobieren lassen. Für Beispiele und Demonstrationen kann das praktisch sein, weil man hier keine `main()`-Methode benötigt. So uneingeschränkt positiv gilt das allerdings nur, wenn man Dinge aus dem Modul `java.base` nutzt. Ansonsten ist in der Regel ein zusätzlicher Import in der JShell notwendig. Da sollte man schon recht genau wissen, aus welchem Package eine Klasse stammt. Bereits hier profitiert man von den Hilfen einer IDE.

6.2.4 Modifikationen in der Versionierung

Nachdem mit Java 9 endlich ein sinnvolles, nachvollziehbares Versionsschema existiert und zudem die unterstützende Klasse `java.lang.Runtime.Version` in das JDK aufgenommen wurde, ist das Ganze aufgrund einer geänderten Releasepolitik von Oracle leider mit Java 10 bereits teilweise wieder über der Haufen geworfen worden. Man spricht nun nicht mehr von Major- und Minor-Version sowie Security-Patch mit dem Format MAJOR.MINOR.SECURITY, sondern von Feature, Interim, Update und Patch mit dem Format FEATURE.INTERIM.UPDATE.PATCH. Deswegen sind die Methoden `major()`, `minor()` und `security()` seit Java 10 deprecated und rufen die Methoden `feature()` bzw. `interim()` sowie `update()` auf.

Beispiele

Betrachten wir ein Beispiel zur Demonstration der Neuerungen:

```
var version = Runtime.version();

System.out.println(version);
System.out.println("Major: " + version.major());
System.out.println("Minor: " + version.minor());
System.out.println("Security: " + version.security());
System.out.println("Feature: " + version.feature());
System.out.println("Interim: " + version.interim());
System.out.println("Update: " + version.update());
System.out.println("Patch: " + version.patch());
System.out.println("Build: " + version.build());
```

Als Ausgabe erhält man Folgendes:

```
10.0.2+13
Major: 10
Minor: 0
Security: 2
Feature: 10
Interim: 0
Update: 2
Patch: 0
Build: Optional[13]
```

Man sieht sehr schön, dass sowohl `major()` und `feature()` sowie `minor()` und `interim()` als auch `security()` und `update()` jeweils gleiche Werte liefern.

Allerdings wirkt das neue API durch die Vielzahl an Methoden schon ein wenig überladen und unhandlich.

Hinweis: Oracles neue Releasepolitik

Bis einschließlich Java 9 wurden neue Java-Versionen immer Feature-basiert veröffentlicht. Das hatte in der Vergangenheit oftmals und mitunter auch beträchtliche Verschiebungen des geplanten Releasetermins zur Folge, wenn ein für die Version wesentliches Feature noch nicht fertig war. Insbesondere deshalb verzögerten sich Java 8 und Java 9 um mehrere Monate bzw. sogar über ein Jahr.

Mit einer zeitbasierten Releasestrategie möchte man dem entgegenwirken. Diese sieht vor, dass jedes halbe Jahr eine neue Java-Version veröffentlicht wird und all jene Features enthält, die bereits fertig sind. Alle drei Jahre ist dann eine sogenannte LTS-Version (Long Term Support) geplant. Eine solche ist in etwa vergleichbar mit den früheren Major-Versionen. Zumindest bezüglich der halbjährlichen Releases hat Oracle schon Wort gehalten, wenn auch Java 9 etwas überraschend nicht als LTS ausgelegt war. Schauen wir gespannt in die Zukunft.

6.2.5 Verschiedenes

In weiteren Bereichen des JDKs gibt es einige kleinere Neuerungen. Diverse davon sind jedoch eher selten für uns als Applikationsentwickler von Relevanz. Zwei potenziell recht interessante sind Folgende:

- `long transferTo(Writer)` aus der Klasse `java.io.Reader`: Es werden alle Zeichen aus dem `Reader` in den übergebenen `Writer` übertragen – diese Funktionalität existiert analog in der Klasse `java.io.InputStream` seit Java 9.

```
var sr = new StringReader("Hello");
var sw = new StringWriter();

sr.transferTo(sw);

System.out.println("SW: " + sw.toString());
```

  Kopieren Sie diese Zeilen in die JShell oder betten Sie diese in eine `main()`-Methode ein, um folgende Ausgabe zu erhalten:

```
SW: Hello
```

- `String toString(Charset)` aus der Klasse `java.io.ByteArrayOutputStream`: Das ist eine überladene Variante der `toString()`-Methode, die das übergebene `Charset` verwendet. Im nachfolgenden Beispiel definieren wir die vier Buchstaben A, B, C und D als ASCII-Codierung in Form eines Byte-Arrays. Im Anschluss verarbeiten wir sie dann mit den passenden Streams wie folgt:

```
var values = new byte[]{'A', 'B', 'C', 'D'};
var is = new ByteArrayInputStream(values);
var bos = new ByteArrayOutputStream();

is.transferTo(bos);

System.out.println(bos.toString());
System.out.println(bos.toString(StandardCharsets.UTF_16));
```

  Beim Ausprobieren des obigen Sourcecodes erhält man zunächst die vier Buchstaben als Ausgabe. Wechselt man das `Charset` in UTF-16, so ändert sich die Ausgabe in chinesische Schriftzeichen. Das ist nachfolgend in Abbildung 6-2 angedeutet:

ABCD
䅂䍄

Abbildung 6-2 Ausgaben der Schriftzeichen mit unterschiedlichem Charset

Hinweis: Imports in der JShell

Zum Ausprobieren dieses Beispiels in der JShell ist ein passender Import etwa wie folgt erforderlich, weil die zur Angabe der Zeichensatzcodierung verwendete Klasse `java.nio.charset.StandardCharsets` nicht im Modul `java.base` vorhanden ist.

```
jshell> import java.nio.charset.*;
```

Wissenswertes

Schließlich möchte ich noch folgende Dinge erwähnen:

- Die mit Java 9 eingeführte JShell besaß recht lange Startup- und Teardown-Zeiten, was dem Gedanken des schnellen Ausprobierens doch im Weg stand. Mit Java 10 erfolgen der Start und auch das Beenden der JShell schneller.
- Die JVM ist seit Java 10 Docker-aware. Das bedeutet, dass die JVM nun erkennt, wenn sie in einem Docker-Container läuft. Dadurch werden spezifische Informationen wie die Anzahl an CPUs, RAM usw. nicht mehr vom Betriebssystem, sondern vom Docker-Container ermittelt.

6.3 Fazit

Java 10 ist eher ein kosmetisches Release und hätte meines Erachtens auch genauso gut die Versionsnummer 9.1 oder 9.2 tragen können – über die neue Releasepolitik von Oracle lässt sich kontrovers diskutieren: Änderungen schneller veröffentlichen zu können, ist sicher positiv. Jedoch dafür immer die Major-Version hochzuzählen, gleichzeitig aber kaum relevante Neuerungen zu integrieren, kann man infrage stellen.

Dies außer Acht lassend rekapitulieren wir nochmals die wesentlichen Features aus Java 10:

1. `var`: Hiermit ist es für lokale Variablen möglich, auf die konkrete Typangabe zu verzichten: Dadurch wird der Sourcecode etwas kürzer, vor allem dann, wenn man Variablen nutzt, die viel Generics verwenden.
2. `copyOf()` und `toUnmodifiableList/Set/Map()`: Diese Methoden dienen zum Kopieren von Listen, Sets und Maps bzw. zum Bereitstellen unveränderlicher Kopien davon.
3. `orElseThrow()`: Diese Methode sorgt für ein klareres API von `Optional<T>` und löst eine `NoSuchElementException` aus, sofern kein Element vorhanden ist.