8 Tipps zur Migration von Java 7 auf Java 8

Bei der Migration von JDK 7 auf JDK 8 in Real-World-Projekten gibt es einige wenige mir bislang bekannte Fallstricke, die man kennen sollte. Nichtsdestotrotz fällt der Umstieg insgesamt eher leicht und man gewöhnt sich schnell an die Annehmlichkeiten aus Java 8. Insbesondere mithilfe von Lambdas und deren Kombination mit Streams sowie dem Filter-Map-Reduce-Framework lassen sich viele Aufgabenstellungen elegant, präzise und – durch Parallelisierung – unter Umständen sogar performanter lösen.

In Abschnitt 8.1 möchte ich Ihnen ein paar Stolpersteine nennen, auf die ich bei der Migration eines größeren Projekts von JDK 7 auf JDK 8 aufmerksam geworden bin. Abschnitt 8.2 zeigt, wie man durch Nutzung von internen Iterationen diverse Anwendungsfälle vereinfachen kann. Neben diesen nach außen nicht sichtbaren Änderungen im JDK gibt es auch für GUIs gewisse Dinge bei der Migration von JDK 7 auf JDK 8 zu beachten, etwa wie man Swing in JavaFX einbindet bzw. umgekehrt. In Abschnitt 8.3 lernen Sie, wie sich JavaFX und die bisherige GUI-Technologie Swing miteinander kombinieren lassen, sodass eine sanfte Migration von GUIs möglich wird.

8.1 Stolpersteine in den Bibliotheken

Die Unterschiede zwischen JDK 7 und JDK 8 sind in vielen Fällen kaum relevant für eine Migration. Jedoch gilt es zu bedenken, dass es Abweichungen bei der Speicherung und somit Traversierung und Ausgabe von `HashMap<K,V>`s gibt. Auch bei der Ausgabe von Monatsnamen mit einem `SimpleDateFormat` hat sich eine Kleinigkeit verändert. Schließlich ist mir noch eine Korrektur beim Runden von Zahlen mit dem `DecimalFormat` bekannt. All diese Dinge sind allerdings bereits in den Release-Notes von JDK 8 erwähnt, wenn auch vielleicht nicht für jeden offensichtlich. Um Ihnen die Problematik zu verdeutlichen, habe ich das Wesentliche auf einige Beispiele heruntergebrochen.

Wir schauen nachfolgend immer zunächst ein kurzes Programm und dessen Ausgabe mit JDK 7 bzw. JDK 8 an. Danach erläutere ich, wie es zu den Unterschieden kommt.

8.1.1 Verarbeitung mit `HashMaps`

In professionellen Programmen legt man in der Regel viel Wert auf Effizienz und gute Performance. Zur Verwaltung von Schlüssel-Wert-Abbildungen findet man deshalb häufig den Einsatz der Klasse `HashMap<K,V>` zur Datenhaltung. Das ist grundsätzlich eine gute Idee, jedoch sollte man die Voraussetzungen und Implikationen kennen.

Schauen wir uns ein simples Beispiel an, in dem zwei Schlüssel-Wert-Abbildungen `first` $\mapsto$ `firstValue` und `second` $\mapsto$ `secondValue` in einer `HashMap<String, String>` gespeichert werden und diese danach auf der Konsole ausgegeben wird:

```
public static void main(final String[] args)
{
    final Map<String, String> map = new HashMap<>();
    map.put("first", "firstValue");
    map.put("second", "secondValue");

    System.out.println(map);
}
```

Listing 8.1 *Ausführbar als* '**HASHMAPEXAMPLE**'

Führt man das Programm HASHMAPEXAMPLE zunächst einmal mit JDK 7 und dann mit JDK 8 aus, so erlebt man eine Überraschung. Es kommt zu einer unterschiedlichen Reihenfolge in der Ausgabe, nämlich wie folgt:

```
JDK 7: {second=secondValue, first=firstValue}
JDK 8: {first=firstValue, second=secondValue}
```

Wie kommt das?

Eine Speicherung in einer `HashMap<K,V>` basiert auf Werten, die über die Methode `hashCode()` berechnet werden und die eine Art Index in die darunter liegende Hashtabelle bilden. Um eine gleichmäßige Speicherung und performante Zugriffe zu erhalten, sollten für unterschiedliche Objekte auch unterschiedliche Werte beim Aufruf von `hashCode()` berechnet werden. Allerdings ist es durchaus erlaubt, wenn auch nicht sinnvoll, gleiche Werte zu berechnen. Dann kommt es zu sogenannten Kollisionen und die Elemente werden alle einem Bucket zugeordnet und in einer dahinter liegenden Datenstruktur (in Java meistens in einer Liste) gesammelt.

Mit JDK 8 wurde die Speicherung in `HashMap<K,V>`s verändert. Zudem erfolgt eine Optimierung bei Kollisionen. Während bis einschließlich JDK 7 in einem solchen Fall mehrere Werte für einen berechneten Hashwert in einer Liste gespeichert wurden, wird nun ein binärer Baum genutzt, sofern die zu speichernden Werte das Interface `Comparable<T>` erfüllen. Das sorgt dafür, dass bei Kollisionen die nachfolgende Zeit zur Suche von linearer ($O(n)$) auf logarithmische ($O(log(n))$ Komplexität deutlich verringert wird. Unabhängig davon gilt, dass man sich auf eine Reihenfolge innerhalb einer `HashMap<K,V>` nicht verlassen darf. Für weitere Details werfen Sie doch einen Blick in mein Buch »Der Weg zum Java-Profi« [3].

8.1.2 Unterschiede beim Einsatz von `SimpleDateFormat`

Bis zur Einführung des neuen Date And Time API in JDK 8 wurde zuvor oftmals ein `SimpleDateFormat` genutzt, um Datumswerte auszugeben, weil man damit eine recht große Flexibilität und Kontrolle besaß.

Betrachten wir die Ausgabe vom 18.3.2014 als Releasedatum von JDK 8. Dieses wird als `Date` erzeugt – wobei man wieder die Unzulänglichkeiten des alten Datums-APIs erkennt – und dann mit dem Muster `MMM yyyy` mit einer Abkürzung des Monatsnamens und einer vierstelligen Jahreszahl ausgegeben:

```
public static void main(final String[] args)
{
    final SimpleDateFormat sdf = new SimpleDateFormat("MMM yyyy");
    final Date march2014 = new Date(2014 - 1900, 3 - 1, 18);

    System.out.println(sdf.format(march2014));
}
```

Listing 8.2 *Ausführbar als* **'SIMPLEDATEFORMATEXAMPLE'**

Starten wir das Programm SIMPLEDATEFORMATEXAMPLE unter JDK 7 bzw. JDK 8, so kommt es je nach JDK zu unterschiedlichen Ausgaben:

```
JDK 7: Mrz 2014
JDK 8: Mär 2014
```

Wie kommt das?

Bei der Ausgabe von Monatsnamen werden mit JDK 8 zum Teil andere Abkürzungen als mit JDK 7 verwendet, die sich mehr an Standards orientieren. In diesem Rahmen wurde auch die Abkürzung für März von `Mrz` auf `Mär` geändert.

Wahrscheinlich würde das kaum auffallen. Ich habe es auch nur entdeckt, weil einer unserer Unit Tests nach der Umstellung auf Java 8 unerwartet fehlschlug.

8.1.3 Rundung beim `NumberFormat`

Wenn man Gleitkommazahlen mithilfe der Klasse `NumberFormat` verarbeitet, müssen zum Teil Rundungen stattfinden. Schauen wir uns einige Zahlen und deren Ausgabe mit nur einer Nachkommastelle an:

```
public static void main(final String[] args)
{
     final NumberFormat format = NumberFormat.getInstance();
     format.setMaximumFractionDigits(1);

     System.out.println(format.format(1.65));
     System.out.println(format.format(6.75));
     System.out.println(format.format(83.65));
}
```

Listing 8.3 *Ausführbar als* **'NUMBERFORMATEXAMPLE'**