8-geben

Fallstricke in Java 8

Tobias Voß

Die neuen Features in Java 8 vereinfachen an vielen Stellen die Programmierung insbesondere durch die prägnanten Lambda-Ausdrücke. Aber neue Features bringen auch immer neue Fallstricke mit sich, wenn der Code sich doch nicht so intuitiv verhält, wie erwartet. In Form eines Ratespiels, angelehnt an Joshua Blochs Java-Rätsel, deckt dieser Artikel einige Fallstricke auf, denen man im produktiven Code lieber nicht begegnen möchte.

Streams und Lambda-Ausdrücke zur Dateiverarbeitung

In Java 8 sind einige Methoden enthalten, die Streams zur Dateiverarbeitung verwenden. Betrachten wir als Beispiel die Methode `Files.lines`, die eine Datei einliest und den Inhalt als `Stream<String>` zurückliefert, mit einem String pro Eingabezeile. Wir verwenden als Eingabedatei „gs.txt“, eine Liste mit den viadee-Geschäftsstellen Münster und Köln, und ermitteln in Listing 1 die Geschäftsstellen, die mit „M“ beginnen.

```
01 Path path = Paths.get("gs.txt");
02 try (Stream<String> lines = Files.lines(path)) {
03   final Optional<String> first = lines.
04     filter(s -> s.startsWith("M")).findFirst();
05   assertEquals("Münster", first.get());
06 } catch (IOException e) {
07   System.out.println("Datei " + path + "fehlerhaft");
08 }
```

Listing 1: Streams und Lambda-Ausdrücke zur Dateiverarbeitung

Rätsel: Was passiert bei der Ausführung von Listing 1?

- *Antwort A:* Der `assertEquals` ist erfolgreich.
- *Antwort B:* Falls die Datei in ISO-8859-1 codiert ist, wird eine `IOException` geworfen, die in der catch-Klausel abgefangen wird.
- *Antwort C:* Falls die Datei in ISO-8859-1 codiert ist, wird eine `RuntimeException` während der Stream-Verarbeitung (Zeilen 3 und 4 in Listing 1) geworfen.

Bevor wir das Rätsel auflösen, sei darauf hingewiesen, dass `Files.lines` als Standard-Encoding UTF-8 verwendet. Während UTF-8 und ISO-8859-1 bezüglich der ASCII-Zeichen übereinstimmen, sind die deutschen Umlaute unterschiedlich codiert. Damit können wir Antwort A ausschließen, diese ist nur richtig, wenn die Datei in UTF-8 codiert ist. Beim Lesen einer ISO-8859-1-Datei tritt eine `MalformedInputException` bei den Umlauten auf, hier beim „ü“ in Münster. Diese Ausnahme erweitert `IOException`.

Trotzdem ist Antwort C richtig! Die Datei wird „lazy“ gelesen, das heißt, erst in den Zeilen 3 und 4 beim Zugriff auf die Inhalte, genauer gesagt in der `filter`-Methode des Streams. Die entstehende `MalformedInputException` ist eine Checked Exception. Aber die Methode `Stream.filter` hat keine throws-Klausel und erlaubt keine Lambda-Ausdrücke, die Checked Exceptions werfen. Stattdessen wird die Checked Exception gefangen und in einer `RuntimeException` gekapselt, genauer gesagt in einer `UncheckedIOException`. Diese Kapselung in einer `RuntimeException` wird in Java 8 häufig verwendet, um möglichst generische funktionale Interfaces bereitzustellen.

Insbesondere, wenn man die altbekannte `FileReader`-Klasse durch `Files.lines` ersetzt, kann der obige Fehler schnell auftreten, da `FileReader` das lokale Encoding verwendet. Die Angabe eines konkreten Encodings in der Form `Files.lines(path, StandardCharsets.ISO_8859_1)` stellt die korrekte Verarbeitung sicher.

Closures und effektiv final

Es gibt die Möglichkeit, in Lambda-Ausdrücken auf Variablen des umgebenden Gültigkeitsbereichs zuzugreifen. Dieses Konstrukt wird auch mit dem Begriff Closure bezeichnet. Voraussetzung dafür ist, dass diese Variablen effektiv final sind, also nicht verändert werden oder anders ausgedrückt beim Hinzufügen des Schlüsselworts `final` keinen Compiler-Fehler erzeugen.

Listing 2 zeigt ein Beispiel für einen Lambda-Ausdruck, in dem die Schleifenvariable `standort` aus dem umgebenden Gültigkeitsbereich für die `Runnable`-Implementierung benutzt wird. Das Beispiel ist von einer Übungsaufgabe in [Hor14] inspiriert.

```
01 List<Standort> standorte = new ArrayList<>();
02 standorte.add(new Standort("Münster"));
03 standorte.add(new Standort("Köln"));
04 List<Runnable> runners = new ArrayList<>();
05 for (final Standort standort : standorte) {
06   standort.setName(standort.getName().replace("ü", "ue"));
07   runners.add(() -> System.out.println(standort));
08   standort.setName(standort.getName().replace("ö", "oe"));
09 }
10 for (Runnable r : runners) {
11   new Thread(r).start();
12 }
```

Listing 2: Closures und effektiv final

Was passiert bei der Ausführung von Listing 2?

- *Antwort A:* Compiler-Fehler.
- *Antwort B:* Ausgabe von „Muenster“ und „Koeln“ in einer beliebigen Reihenfolge.
- *Antwort C:* Ausgabe von „Muenster“ und „Köln“ in einer beliebigen Reihenfolge.

Das Schlüsselwort `final` in Java dient zur Kennzeichnung von Variablen, die nur einmalig zugewiesen werden, und erzwingt einen Compiler-Fehler bei einer erneuten Zuweisung. Seit Java 8 ist `final` für Lambda-Ausdrücke und innere Klassen optional, das heißt, der Compiler überprüft bei der Verwendung einer Variablen aus dem umgebenden Gültigkeitsbereich auch ohne das Schlüsselwort, ob die Variable effektiv final ist, und er-

zeugt anderenfalls einen Fehler. Für die Schleifenvariable `standort` gilt diese Eigenschaft in jedem Schleifendurchlauf [Sta]. Damit kann Antwort A ausgeschlossen werden.

Die Änderung des Wertes einer Variablen, wie hier in den Zeilen 6 und 8 durch den Aufruf des Setters für den Standortnamen, beeinträchtigt in Java nicht die `final`-Eigenschaft. Java verfügt im Gegensatz zu C++ über keine Möglichkeit, eine Variable als inhaltlich unveränderbar (immutable) zu kennzeichnen oder diese Eigenschaft durch den Compiler zu erzwingen. In diesem Beispiel bleibt die Variable `standort` trotz der Setter-Aufrufe effektiv final.

Aber zu welchem Zeitpunkt wird nun der Wert von `standort` im Lambda-Ausdruck hinterlegt? Die richtige Antwort ist B, das heißt, auch die Ersetzung des ö-Umlauts, die erst nach dem Lambda-Ausdruck steht, wird berücksichtigt. Das liegt daran, dass nur die Referenz auf `standort` in der Closure abgelegt wird. Der Wert wird beim Zugriff ermittelt, und zu dem Zeitpunkt wurde der Name bereits über die Set-Methode verändert.

Stream-Interferenz

Auch für das nächste Beispiel in Listing 3 wird eine Closure verwendet, um mit dem neuen Stream-API die Liste der Standorte aus dem vorherigen Beispiel zu filtern und Standorte mit weniger als fünf Buchstaben zu entfernen.

```
01 Stream<String> standortStream = standorte.stream();
02 standortStream.forEach(s -> {
03   if (s.length() < 5) {
04     standorte.remove(s);
05   }
06 });
```

Listing 3: Stream-Interferenz

Was passiert bei der Ausführung von Listing 3?

- *Antwort A:* Exception.
- *Antwort B:* Compiler-Fehler.
- *Antwort C:* Die `standorte`-Liste enthält nur noch Münster.

Nach den Erläuterungen zum vorherigen Beispiel kann man die Antwort B schnell ausschließen, die Modifikation der `standorte`-Liste ist legal.

Die Streams in Java 8 verfügen über keine eigene Datenhaltung, sondern verwenden die Daten der zugrunde liegenden Collection. Daher verlangt das Stream-API, dass der Lambda-Ausdruck nicht mit der Datenquelle interferiert [Ora], das heißt, die Datenquelle nicht verändert wird. Das gilt sowohl für sequenzielle als auch parallele Streams. In diesem Beispiel verletzt `remove` diese Anforderung. Die Interferenz führt zu einer `ConcurrentModificationException`, damit ist Antwort A richtig.

Der beste Ansatz, um ähnliche Probleme zu vermeiden, ist die volle Ausnutzung des neuen Stream-APIs und der funktionalen Ausdrücke. Für dieses Beispiel eignet sich die `filter`-Methode hervorragend, um das gewünschte Ergebnis in gut lesbarer und korrekter Form zu ermitteln: `standortStream.filter(s -> s.length() >= 5)`.

Auch bei der Verwendung von parallelen Streams, in denen die Ergebnisse nebenläufig ermittelt werden, lassen sich durch die konsequente Verwendung des deklarativen Stream-APIs Synchronisationsprobleme von vorneherein vermeiden. In Listing 4 sind zwei Ansätze zum Zählen von Worten gruppiert nach ihrer Länge aufgeführt. Der erste Ansatz in den Zeilen 1 bis 5 mit einem Array für die Zähler produziert ohne zusätzliche Synchronisation eine Race Condition. Die neue Collect-Methode in den Zeilen 6 bis 8 liefert eine elegante und threadsichere Lösung für dieses Problem.

```
01 final int[] lenCount = new int[10];
02 standorte.stream().parallel().forEach(s -> {
03 if (s.length() < 10)
04   lenCount[s.length()]++; // Race Condition
05 });
06 Map<Integer, Long> lenCount2 = standorte.stream()
07   .parallel().collect(
08     Collectors.groupingBy(String::length, counting()));
```

Listing 4: Paralleler Stream mit und ohne Race Condition

Vererbung bei Default-Methoden

Die bestehenden Java-APIs wurden an vielen Stellen ergänzt, um von den neuen Lambda-Ausdrücken zu profitieren. Das Collection-Interface besitzt beispielsweise seit Java 8 eine `forEach`-Methode, an die ein Lambda-Ausdruck übergeben werden kann, der auf jedes Element der Collection angewendet wird. Diese Methode ist als Default-Methode im Super-Interface `Iterable` implementiert. Default-Methoden sind ein weiteres neues Feature, um generische Methoden in Interfaces zu implementieren. Damit haben die Java-Designer auch eine Möglichkeit geschaffen, Interfaces um neue Methoden zu erweitern, ohne bisherige Implementierungen zu brechen.

In Listing 5 besitzt das Interface `Writable` eine Default-Methode `write`, in der eine konkrete Implementierung direkt im Interface enthalten ist. Darüber hinaus gibt es eine Klasse `Standort`, die `Writable` implementiert, einschließlich der `write`-Methode.

```
01 public interface Writable extends Serializable {
02   default void write(ObjectOutputStream oos)
03       throws IOException {
04     oos.writeObject(this);
05   }
06 }
07 class Standort implements Writable {
08   private String name;
09   // ... weitere Methoden
10   public void write(ObjectOutputStream oos)
11       throws IOException {
12     oos.writeObject(name);
13   }
14 }
```

Listing 5: Vererbung bei Default-Methoden

Was passiert beim Aufruf der `write`-Methode für ein `Standort`-Objekt?

- *Antwort A:* Compiler-Fehler.
- *Antwort B:* Die Methode `Writable.write` wird ausgeführt.
- *Antwort C:* Die Methode `Standort.write` wird ausgeführt.

Die Einführung von Default-Methoden bereichert Java um das Problem der Mehrfachvererbung, das man bei Klassen explizit ausgeschlossen hat. Der Compiler prüft darauf und wirft einen Fehler, falls eine Klasse zwei Interfaces implementiert, die eine Default-Methode mit derselben Signatur bereitstellen. Um das Problem zu lösen, muss die Klasse diese Methode implementieren.

In Listing 5 überschreibt die Klasse `Standort` die `write`-Methode. Damit tritt kein Compiler-Fehler auf, selbst wenn `Standort` ein weiteres Interface mit einer `write`-Methode implementieren würde. Für Java gilt in dem Fall die „class wins"-Regel, die besagt, dass die Implementierung in der Klasse immer der Default-Methode aus dem Interface vorzuziehen ist. Damit ist Antwort C korrekt.

Diese Regel betrifft auch Methoden aus der Klasse `Object`. Eine Implementierung einer Default-`toString`-Methode in einem Interface kann also nie ausgeführt werden und wird bereits vom Compiler als Fehler markiert.

Umgang mit Unsigned-Typen

Die ganzzahligen Wrapper-Typen unterstützen nun vorzeichenlose Arithmetik, wie man sie aus den Unsigned-Typen in C++ kennt. Der primäre Grund für diese Erweiterung ist die Möglichkeit, mit Dateiformaten und Netzwerkprotokollen umzugehen, die Unsigned-Typen verwenden. Die Methode `toUnsignedInt` interpretiert einen vorzeichenbehafteten Byte- oder Short-Wert als einen vorzeichenlosen Integer-Wert.

```
01 byte b1 = -25;
02 byte b2 = 24;
03 int i1 = Byte.toUnsignedInt(b1);
04 int i2 = Byte.toUnsignedInt(b2);
05 assertTrue(i1<i2);
```

Listing 6: Umgang mit Unsigned-Typen

Was passiert bei der Ausführung von Listing 6?

- *Antwort A:* Der `assertTrue` schlägt fehl, da `i1`=25 ist und damit größer als `i2`=24.
- *Antwort B:* Der `assertTrue` ist erfolgreich.
- *Antwort C:* Der `assertTrue` schlägt fehl, auch falls `b1` zwischen -1 und -23 ist.

Für C++-Programmierer ist diese Frage vermutlich einfach zu beantworten, da diese mit der Unsigned-Problematik im Zusammenhang mit der binären Repräsentation von Zahlen vertraut sind. Negative Zahlen werden binär als sogenanntes Zweierkomplement [Wiki] abgebildet. Die Zahl -25 wird zuerst ohne Vorzeichen in ihren Binär-Wert 0001 1001 umgerechnet, dann invertiert beziehungsweise negiert in den Wert 1110 0110 und nach der Addition von 1 ergibt sich die Zweierkomplement-Darstellung 1110 0111, die einem Unsigned-Wert von +231 entspricht. Als einfache Umrechnungsregel gilt: Zum vorzeichenbehafteten Wert muss immer 2^8=256 addiert werden, um die vorzeichenlose Interpretation zu errechnen.

Die Methode `toUnsignedInt` führt also keine Konvertierung im Sinne des Absolutbetrags durch, sondern interpretiert das Byte als vorzeichenlose Zahl statt wie üblich im Zweierkomplement.

Damit ergibt sich C als richtige Antwort. Der `assertTrue` schlägt fehl, da +231 nicht kleiner als +24 ist. Dasselbe Ergebnis würde bei `b1`=-1 und damit `i1`=+255 eintreten.

Zur Vereinfachung des Umgangs mit Unsigned-Werten kann man auch direkt die Methoden `compareUnsigned`, `divideUnsigned` und `remainderUnsigned` statt der im Beispiel gezeigten Konvertierung verwenden. So liefert `Integer.compareUnsigned(b1,b2)` einen Wert kleiner 0 zurück, weil das Byte `b1` als vorzeichenloser Wert +231 interpretiert größer als `b2` ist. Für die Addition, Subtraktion und Multiplikation gibt es keine speziellen Methoden, da diese unabhängig von der Interpretation als vorzeichenloser oder -behafteter Wert binär dasselbe Ergebnis liefern.

Ablösung der Calendar-Klasse durch LocalDate

Das neue Date-/Time-API ist ein großer Fortschritt in der Java-API-Familie. Es ersetzt die bisherigen Klassen `Date` und `Calendar` und macht auch die Nutzung von Dritt-Bibliotheken wie JodaTime [Joda] überflüssig. Im stark vereinfachten Listing 7 hat ein Entwickler die `Calendar`-Klasse durch `LocalDate` ersetzt und dabei direkt einige Zeilen eingespart, da die Setter-Methoden entfallen sind. Die Klasse `LocalDate` ist immutable und hat deswegen keine Setter und der Wert muss im Konstruktor angegeben werden. Durch die Unveränderlichkeit umgeht das neue API einige Probleme, die bei Multi-Threading entstehen können.

```
01 // final Calendar cal = Calendar.getInstance();
02 // cal.set(Calendar.YEAR, 2014);
03 // cal.set(Calendar.MONTH, Calendar.JULY);
04 // cal.set(Calendar.DATE, 18);
05 // int day = cal.get(Calendar.DAY_OF_WEEK);
06 int day = LocalDate.of(2014, Calendar.JULY, 18)
07                    .getDayOfWeek().getValue();
08 switch (day) {
09 case Calendar.FRIDAY:
10   System.out.println("Thank God, it's Friday");
11   break;
12 }
```

Listing 7: Ablösung der Calendar-Klasse durch LocalDate

Was passiert bei der Ausführung von Listing 7, angenommen für die restlichen Wochentage wird im Switch jeweils der Name des Tages ausgegeben?

- *Antwort A:* Ausgabe: „Thank God, it's Friday".
- *Antwort B:* Ausgabe: „Donnerstag".
- *Antwort C:* Ausgabe: „Dienstag".

Bei der Ersetzung der `Calendar`-Klasse durch `LocalDate` im Beispiel wurden zwei Implementierungsdetails der jeweiligen Klassen nicht berücksichtigt. In der `Calendar`-Klasse ist der Sonntag gemäß amerikanischem Verständnis der erste Tag der Woche: Die Konstante `Calendar.SUNDAY` hat den Wert eins, die Tage Montag bis Samstag die Werte zwei bis sieben. Im Enum `DayOfWeek` des neuen APIs orientiert man sich an der Norm ISO 8601, in der gemäß europäischer Lesart der Montag als erster Wochentag definiert ist. Das ist aber nicht das einzige Problem, das auftritt, denn mit dieser Erklärung wäre Antwort B richtig: `DayOfWeek.FRIDAY=5` entspricht `Calendar.THURSDAY`.

Darüber hinaus wird bei `LocalDate` der Monat numerisch beginnend mit eins angegeben, wie man es vom Kalendermonat erwartet, also Juli=7. Die Konstanten aus `Calendar` für die Monate beginnen hingegen bei null, das heißt, `Calendar.JULY=6`.

Tatsächlich ist im Beispiel `day=18.06.2014`. Das ist ein Mittwoch, und aus `DayOfWeek.WEDNESDAY=3` wird beim Switch über die Calendar-Konstanten `Calendar.TUESDAY=3` und damit ist die Antwort C richtig.

Es ist empfehlenswert, stattdessen direkt das Enum `DayOfWeek` zu verwenden und darüber die Switch-Anweisung zu bilden. Im Beispiel tritt durch die Vermischung von `LocalDate` und den `Calendar`-Konstanten ein Fehler auf. In diesem reduzierten Beispiel ist das ziemlich offensichtlich, aber in realen Projekten versteckt sich dieser Fehler bestimmt besser über verschiedene Klassen und Architekturschichten hinweg.

Fazit

Die harmlos aussehenden Beispiele demonstrieren einige Fallstricke bei der Benutzung der neuen Java 8-Features. Trotz des erhöhten Abstraktionsgrades durch die neuen Lambda-Ausdrücke muss man sich im Einzelfall immer wieder mit Implementierungsdetails des APIs beschäftigen. Auch bei den anderen neuen Features lauern versteckte Gefahren bei einer unbedachten Nutzung.

Literatur und Links

[Blo05] J. Bloch, N. Gafter, Java Puzzlers, Traps, Pitfalls, and Corner Cases, Addison-Wesley, 2005
[Hor14] C. S. Horstmann, Java SE 8 for the Really Impatient, Addison-Wesley, 2014
[Joda] Joda-Time, `http://www.joda.org/joda-time/`
[Ora] Oracle, Java SE 8 API Specification, `http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html`
[Sta] Stack Overflow, `http://stackoverflow.com/questions/3911167/how-does-final-int-i-work-inside-of-a-java-for-loop`
[Wiki] Wikipedia, `http://de.wikipedia.org/wiki/Zweierkomplement`

Tobias Voß ist Softwareentwickler und Projektleiter bei der viadee IT-Unternehmensberatung. Als Berater begleitet er Kunden bei der Umsetzung von individuellen Softwaresystemen auf der Basis von Java EE oder Mainframe-Architekturen.
E-Mail: tobias.voss@viadee.de