Besserer Java-Code – außerhalb der Automatismen

Wolfgang Nast, MT AG

Mit jeder neuen Java-Version kamen auch Verbesserungen in der Art und Weise, wie Code besser geschrieben werden kann. Wenn man nützliche Beispiele gesehen hat, unterstützt einen dabei die Entwicklungsumgebung.

Die wichtigste Unterstützung beim Codieren sind Entwicklungsumgebungen (IDE) wie Eclipse, IntelliJ und NetBeans. Dabei sollte in einem Projekt eine einheitliche Vorgabe beim Formatieren und Überprüfen eingestellt sein.

Verbreitete Tools zum Finden und Vermeiden von unschönem Code sind Checkstyle, FindBugs und SonarQube. Diese können sowohl im automatischen Build mit Maven oder Gradle eingebunden sein als auch in die IDEs selbst, damit man schon beim Codieren unterstützt wird. Nun sind die üblichen Automatismen ausreichend beschrieben und wir kommen zum Teil der Verbesserungen.

Es geht um Neuerungen von Java 5 bis 9, die meist zusammen vorgestellt werden. Mit dabei sind Themen wie Exception Handling, Generics, Collections mit Sortierungen, Streams mit Primitiven und optionale Werte (null). Im Artikel sind die Verbesserungen in zwei bis drei Blöcken beschrieben. Dabei ist im ersten Block der Code, der bisher üblich war, danach kommt der verbesserte Code. Im optionalen dritten Block ist der praktische Hilfscode beschrieben.

Beginnen wir mit den Ressourcen (wie „IOStreams"), die nach der Verwendung wieder geschlossen werden müssen. Bis Java 7 war folgender Code üblich *(siehe Listing 1)*. Hier lässt sich mit „try with resources" viel Code sparen *(siehe Listing 2)*.

Dabei entfällt der rot markierte „finally"-Block und die Fehlerbehandlung geschieht an einer Stelle. Die Variable „inStream" ist nur im Block gültig. Dies kann ab Java 9 wieder erweitert werden, wenn diese implizit final ist und vor dem Block angelegt wird. Bei dem Code in *Listing 3* möchte man gerne von der vorgestellten Ressourcen-Verwendung profitieren. Auf den „finally"-Block kann verzichtet werden, wenn die Lock-Verwendung zu einer Ressource wird *(siehe Listing 4)*. Hier steht ein „try"-Block ohne „catch" und ohne „finally", aber mit Ressource. Die Hilfsklasse „LockBlock" ist in *Listing 5* beschrieben.

Der blau markierte Code ist der Teil, der von der Ressource ausgeführt wird; der grüne Code ist von der Ressource vorgegeben. Der „close"-Teil wird beim Verlassen des Blocks aufgerufen. Zu beachten ist, dass der „throws Exception"-Teil, der vom Interface „Closeable" vorgegeben wird, weglassen wurde.

Im Konstruktor ist der Teil des Holens der Ressource übernommen. Die Hilfsklasse hat zwei Konstruktoren; der zweite dient dazu, den Lock mit der Methode „lockInterruptibly" zu holen, der auch eine „InterruptedException" werfen kann. Der zweite Parameter „inter" wird nicht ausgewertet, da er nur dazu dient, eine andere Signatur zu haben, weil der „throws"-Teil nicht zur Unterscheidung dient, obwohl dieser beim Aufruf wichtig ist. Es folgt das Beispiel mit der Verwendung der „InterruptedException" *(siehe Listing 6)*. Die ursprüngliche Collection-Klasse „Vector" wurde wie in *Listing 7* verwendet.

Hier ist in Blau der Teil mit der Initialisierung und in Orange der Teil mit dem Ermitteln der grünen Werte dargestellt. Es lässt sich übersichtlicher codieren *(siehe Listing 8)*. Die Initialisierung in Blau kommt mit Java 9. Der rote Code sollte anstelle des Fragezeichens den verwendeten Typ angeben. Die ursprüngliche Hashtable-Klasse zum Mappen von Werten sah wie in *Listing 9* aus. Auch hier ist die Initialisierung blau und der Teil zum Ermitteln der grünen Werte orange. Auch die Map lässt sich übersichtlicher codieren *(siehe Listing 10)*.

```
InputStream inStream = null;
try {
     inStream = new FileInputStream (Datei);
//Logic
  } catch (IOException e) {
     //Fehlerbehandlung 1
} finally {
  try {
     if (inStream != null) {
        instream.close();
     }
  }catch (IOException e) {
     //Fehlerbehandlung 2
  }
}
```

Listing 1

```
  try (InputStream inStream = new
FileInputstream(Datei)) {
     //Logic
  } catch (IOException e){
     //Fehlerbehandlung 1 und 2
  }
```

Listing 2

```
  Lock lock = new ReentrantLock();
  try {
     lock.lock();
     //Logic;
  } finally{
     lock.unlock();
  }
```

Listing 3

```
  Lock lock = new ReentrantLock();
  try (LockRes bl = new LockBlock(lock)){
     //Logik;
     return Ergebnis;
  }
```

Listing 4

```
public class LockRes() implements Closeable{
  private Lock lock;
  public LockRes(Lock lock) {
  this.lock = lock;
     lock.lock();
  }
  public LockRes(Lock lock, boolean inter) throws
           InterruptedException {
     this.lock = lock;
     lock.lockInterruptibly();
  }
  @Override
  public void close() {
     lock.unlock();
  }
 }
```

Listing 5

```
  Lock lock = new ReentrantLock();
  try (LockRes bl = new LockBlock(lock, true)){
     //Logic
  } catch(InterruptedException e) {
     //Unterbrochen
  }
```

Listing 6

Bei der Initialisierung der Werte ist zu beachten, dass bis zu zehn Werte-Paare nacheinander angegeben werden können. Bei elf oder mehr Werten ist die Methode „Map.ofEntries(Map.Entry<K,V>...entries)" zu verwenden. Bei Streams von „int"-Werten ist recht häufig der Code aus *Listing 11* zu finden.

Im blauen Teil wird der Objekt-Datentyp von „int" verwendet, was zu Autoboxing und Autounboxing führt. Der rote Teil wird dabei gerne weggelassen, weil es auch ohne die „Null"-Tests meistens gut geht. Die Methode „mapToInt" mit dem Identity-Aufruf „x -> x" funktioniert dank Autounboxing. Besser ist hier *Listing 12*. Doch es geht auch ohne Autoboxing *(siehe Listing 13)*.

Es gibt nur für die primitiven Datentypen „int", „long" und „double" passende Streams. Bei „float" ist auf „double" zu erweitern, „byte", „char" sowie „short" auf „int". Beim Verwenden von optionalen Werten wird meist „null" als „nicht vorhanden" gekennzeichnet. Damit sind viele „null"-Tests verbunden, wie *Listing 14* zeigt. Hier eignet sich die Klasse „Optional" besser *(siehe Listing 15)*.

Dabei wird durch „map" zweimal der „null"-Test ausgeführt und ein neues „Optional" angelegt. Mit „ifPresent" wird nur der „Logic"-Block aufgerufen, wenn „Optional" gesetzt ist (nicht „null"). Optional gibt es auch die primitiven Datentypen „int", „long" und „double". Sie eignen sich besser, um optionale Werte darzustellen, als die Objekt-Entsprechungen. Wie die Objekt-Entsprechungen ist „Optional" ein konstanter Wert; bei dessen Änderungen ist eine neue Instanz anzulegen. In Java gibt es keinen „call by reference", deshalb wird für primitive Datentypen und konstante Werte gerne der Ansatz aus *Listing 16* verwendet.

Rot markiert sind hier die Änderungen, die durch die Verwendung des Arrays mit einem Element entstanden sind. Dies ist bei interner Nutzung ausreichend. Bei der Verwendung als API muss geprüft werden, ob das Array nicht „null" ist und es nur die Größe „eins" hat. Bei der Verwendung einer lokalen Variablen in der „For"-Schleife ist es direkt möglich *(siehe Listing 17)*. *Listing 18* zeigt, wie man die „For"-Schleife auf eine Lambda-Expression in der Methode „forEach" umstellt.

Da alle lokalen Variablen in der Lambda-Expression „final" sein müssen, ist die Änderung in Rot notwendig. Es kann zwar das „final" weggelassen werden, der Compiler setzt es jedoch implizit. Meistens lassen sich die Lambda-Blöcke soweit optimieren, dass keine

```
Vector werte = new Vector();
  werte.addElement(wert1);
  werte.addElement(wert2);
for(int i = 0; i <werte.size(); ++i) {
  Object wert = werte.get(i);
  //Logic
}
```

Listing 7

```
Collection<?> werte = List.of(wert1, wert2);
werte.forEach(wert -> {
  //Logic
});
```

Listing 8

```
Hashtable map = new Hashtable();
map.put(key1, wert1);
map.put(key2, wert2);
for(Enumeration keyEnum = map.keys(); keyEnum.hasMo-
reElements(); ) {
  Object key = keyEnum.nextElement();
  Object wert = map.get(key);
  //Logik
}
```

Listing 9

```
Map<?, ?> map = Map.of(key1, wert1,
  key2, wert2);
map.forEach((key, wert) -> {
  //Logik
});
```

Listing 10

```
Collection<Integer> zahlen = List.of(1,3,6);
zahlen.stream().filter(x -> x != null)
  .mapToInt(x -> x)
  .forEach(System.out::println);
```

Listing 11

```
Collection<Integer> zahlen = List.of(1,3,6);
zahlen.stream().filter(Objects::nonNull)
  .mapToInt(Integer::intValue)
  .forEach(System.out::println);
```

Listing 12

```
IntStream zahlenSt = IntStream.of(1,3,6);
zahlenSt.forEach(System.out::println);
```

Listing 13

```
if (wert != null) {
  if (wert.getText() != null) {
     if (wert.getText().getTeile() != null) {
        //Logic
     }
  }
}
```

Listing 14

```
Optional.ofNullable(wert)
  .map(Wert::getText)
  .map(Text::getTeile)
  .ifPresent(x -> {
     //Logic
});
```

Listing 15

```
public void aufruf(int[] ref){
  ref[0]++;
}
int zahlRef[] = {0};
aufruf(zahlRef);
```

Listing 16

```
boolean gefunden = false;
for(Wert wert : wertList) {
  //Logic
  gefunden = true;
}
```

Listing 17

```
final boolean gefunden[] = {false};
wertList.forEach(wert -> {
  //Logic
  gefunden[0] = true;
});
```

Listing 18

```
public static Comparator<WerteKlasse> comp =
  Comparator.comparingLong(WerteKlasse::getId)
  .thenComparing(WerteKlasse::getText);
```

Listing 19

```
public static Comparator<WerteKlasse> compNull =
  Comparator.nullsFirst(comp);
```

Listing 20

```
@Override
public int compareTo(WerteKlasse o) {
  return comp.compare(this, o);
}
```

Listing 21

```
@Override
public boolean equals(Object obj) {
  if (this == obj) {
     return true;
  }
  if (obj != null && !getClass().equals(obj.get-
Class())) {
     return false;
  }
  return compareTo((WerteKlasse)obj) == 0;
}
```

Listing 22

```
@Override
public int hashCode() {
  return Objects.hash(id, text);
}
```

Listing 23

```
public void aufruf(Wert wert) {
  if (wert == null) {
     throw new NullPointerException(msg);
  }
  //Logic
}
```

Listing 24

lokalen Variablen verwendet werden müssen. Bei der Verwendung von eigenen Klassen in Collections kommt es vor, dass diese auch sortiert werden sollen; dafür gibt es den „Comparator". Dieser lässt sich mittlerweile wie in *Listing 19* realisieren.

Hier wird in der Reihenfolge der gewünschten Sortierung der Getter der Werte angegeben. Sollen auch „null"-Werte mit sortiert werden, lässt sich dieser zweite Comparator wie in *Listing 20* angegeben. Alternativ geht auch „nullsLast". Für die natürliche Sortierung der Klasse ist das Interface „Compareable<Klasse>" zu implementieren. Dies kann recht leicht mit den Comparator geschehen *(siehe Listing 21)*.

Leider wird häufig vergessen, dass auch „equal" und „hasCode" zu überschreiben sind, um eine natürliche Sortierung richtig umzusetzen. *Listing 22* zeigt den Code für „equals".

Der rote Code ist notwendig, um auf den Parameter „Object" richtig zu reagieren. Wichtig ist, dass die Klasse gleich sein muss, damit „a.equals(b) == b.equals(a)" für alle „a" und „b" gilt. Der Test „instanceof" ist nicht ausreichend. *Listing 23* zeigt den Code für „hasCode".

Die Methode „Objects.hash(Object ... objs)" beachtet auch „null"-Werte beim Berechnen des Hash-Werts. Bei Parametern in Schnitt-

```
public void aufruf(Wert wert) {
  Objects.requireNonNull(wert, msg);
     //Logic
}
```

Listing 25

```
public Object update(Updateable daten){
  try {
     return em.update(daten);
  }
  return null;
}
daten1 = (Daten1)update(daten1);
daten2 = (Daten2)update(daten2);
```

Listing 26

```
public <T extends Updateable>
  T update(T daten){
  try {
     return em.update(daten);
  }
  return null;
}
daten1 = update(daten1);
daten2 = update(daten2);
```

Listing 27

```
@SuppressWarnings("unchecked")
public <T> T aufruf(int pos){
  return (T)daten[pos];
}
Daten1 daten1 = aufruf(1);
Daten2 daten2 = aufruf(2);
```

Listing 28

stellen sind meistens „null"-Tests notwendig. *Listing 24* zeigt die übliche Umsetzung. Dabei ist der zu testende Wert grün markiert. Hier bietet die Klasse „Objects" eine kürzere Schreibweise *(siehe Listing 25)*.

Der „Null"-Test und das „Throw"-Statement sind in der Methode „requireNonNull" umgesetzt, es muss noch die Message übergeben werden, wenn der Wert „null" ist. Beim Aufruf von Speicher-Methoden wird meist das gespeicherte Objekt wieder zurückgegeben, was zu einem Cast bei der Verwendung führt *(siehe Listing 26)*. Mit einem generischen Parameter kann auf den roten Cast verzichtet werden, damit ist der Code wie in *Listing 27*.

```
public class Werte {
  private final long id;
  private final String name;
  private final String vorname;
  public Werte(long id, String name, String vorname) {
     this.id = id;
     this.name = name;
     this.vorname = vorname;
  }
  public Werte(long id, String name) {
     this(id, name, null);
  }
  public long getId() {
     return id;
  }
  public String getName() {
     return name;
  }
  public String getVorname() {
     return vorname;
  }
}
```

Listing 29

```
public class WerteBuilder {
  private long id = 0;
  private String name;
  private String vorname;
  public WerteBuilder() {
  }
  public WerteBuilder withId(long id) {
     this.id = id;
     return this;
  }
  public WerteBuilder withName(String name) {
     this.name = name;
     return this;
  }
  public WerteBuilder withVorname(String vorname) {
     this.vorname = vorname;
     return this;
  }
  public Werte build() {
     return new Werte(id, name, vorname);
  }
}
```

Listing 30

```
Wert wert1 = new Wert(12, "Müller", "Karl");
Wert wert1 = new WerteBuilder()
  .withId(12)
  .withVorname("Karl")
  .withName("Müller").build();
```

Listing 31

Hier bestimmt der Typ des Parameters den Typ des Rückgabewerts. Der „Updateable"-Typ, der für den Aufruf notwendig ist, steht jetzt im generischen Parameter nach dem „Extends". Es ist auch möglich, den Cast des Rückgabewerts in die Methode zu verlagern, damit er nicht nach dem Aufruf notwendig wird *(siehe Listing 28)*. Hier sollte die Warnung des Compilers an der orangenen Stelle mit der Annotation „SuppressWarnings()" auch in Orange unterdrückt werden.

Gerne werden konstante Container-Klassen wie in *Listing 29* codiert. Dabei sind die angebotenen Konstruktoren immer unpraktischer, je mehr Werte übergeben werden müssen und wenn mehrere Werte optional sind, wie im Beispiel mit Vorname. Hier eignen sich „Builder"; diese kommen nicht durch Neuerungen in Java, sondern aus der Verbesserung von APIs *(siehe Listing 30)*.

Der Builder ist keine Vereinfachung des zu implementierenden Codes, da er komplett zusätzlich zur Klasse kommt. Er ist sogar umfangreicher als die eigentliche Klasse, da alle „Setter(with)" sich „Selbst(this)" zurückgeben, und er eine Build-Methode zum eigentlichen Anlegen der Instanz in Grün hat. In Ocker sind die Erweiterung des Builders und die Konstruktoren angegeben. Erst beim Aufruf zeigt der Builder seinen Vorteil *(siehe Listing 31)*.

Hier ist beim Builder über die blauen „with"-Methoden immer klar, welcher Wert gesetzt wird, auch wenn sich die Anzahl der Parameter mal erweitern sollte. Der Aufrufer bestimmt die Reihenfolge. Beim Konstruktor muss man bei vielen Parametern vom gleichen Typ aufpassen (Name oder Vorname zuerst).

Fazit

Es sind viele Verbesserungen mit den neuen Versionen von Java gekommen, die auch einfacheren Code ermöglichen. Streams, Optional und Builder werden gerne im Code-Chaining verwendet. Das heißt, das Ergebnis wird für den nächsten Aufruf verwendet und nicht mehr in einer Variablen zwischengespeichert. Man hat damit eine Kette von Aufrufen.

Wolfgang Nast
wolfgang.nast@mt-ag.com

Wolfgang Nast (Dipl. Ing.) ist Senior Berater bei der MT AG und seit dem Jahr 1998 mit Java SE sowie seit dem Jahr 2006 mit Java EE in den Bereichen „Schnittstellen" und „Architektur" beratend und umsetzend tätig.