Fehler und Korrekturen

zum Buch "Java ist eine Sprache"

von Ulrich Grude.

Eine gute Nachricht: Mehrere Seiten im Buch "Java ist eine Sprache" scheinen keine schwerwiegenden Fehler zu enthalten (z. B. die Seiten VII bis XII, die automatisch erzeugt wurden :-).

Noch eine gute Nachricht: Einige schwerwiegende Fehler auf anderen Seiten sind bereits bekannt geworden und im Folgenden findet man Korrekturen dafür.

Eine schlechte Nachricht: Vermutlich enthält das Buch noch weitere Fehler. Hinweise auf diese kleinen und großen noch Unbekannten sind jederzeit willkommen (e-mail: grude@beuth-hochschule.de).

Schreibfehler, deren Korrektur einfach und offensichtlich ist, werden hier *nicht* erwähnt.

Erläuterung:
Eine Stellenangabe wie z. B. S. 15, Z. 3 bezeichnet die Zeile 3 auf der Seite 5. Bei einer Stellenangabe mit negativer Zeilennummer, z. B. S. 22, Z. -7, sind die Zeilen *von unten nach oben* zu zählen.

Diese Fehlerliste wurde zuletzt ergänzt am 10.11.2010
Diese Fehlerliste wurde zuletzt ergänzt am 30.08.2007
Diese Fehlerliste wurde zuletzt ergänzt am 08.05.2007
Diese Fehlerliste wurde zuletzt ergänzt am 20.02.2007
Diese Fehlerliste wurde zuletzt ergänzt am 10.07.2006
Diese Fehlerliste wurde zuletzt ergänzt am 24.01.2006
Diese Fehlerliste wurde zuletzt ergänzt am 15.01.2006
Diese Fehlerliste wurde ergänzt am 24.11.2005

S. 8, Z. 18: Statt "(Anfang 2002)" sollte es "(Anfang 2005)" heißen.

S. 25, Z. -6: Statt "1.4.1" sollte es "1.5.1" heißen.

S. 34, Z. -6: Diese Zeile ist mit 16 nummeriert. Statt `summe = summe + n;` sollte es `summe = summe + ein;` heißen.

S. 106, Z. -13: In der Float-Regel-03 sollte es 2^{24} - 2 heißen (statt 2^{24} - 1).

S. 106, Z. -3: In der Double-Regel-03 sollte es 2^{53} - 2 heißen (statt 2^{53} - 1).

S. 109, Z. -4: Hier sollte es heißen:
Eine nicht-normalisierte `float` Zahl hat den Exponenten -127 und den Binärpunkt sollte man sich zwischen dem ersten und zweiten Bit der 23-Bit-Mantisse vorstellen (d.h. zwischen Bit 22 und Bit 21 des `float`-Wertes).

Eine normalisierte `double`-Zahl hat einen Exponenten zwischen -1022 und +1023 [nicht: zwischen -2046 und +2047] und eine Mantisse der Form 0.xxx... mit 52 (binären) Nachpunktstellen

S. 110, Z. 1: Hier sollte es heißen:
Eine nicht-normalisierte `double`-Zahl hat den Exponenten -1023 [nicht: -2047] und den Binärpunkt sollte man sich zwischen dem ersten und zweiten Bit der 52-Bit-Mantisse vorstellen (d.h. zwischen Bit 51 und 50 des `double`-Wertes).

S. 140, Z. 15: Statt "und neun Konkatenationsoperationen" sollte es "und 18 Konkatenationsoperationen" heißen. Begründunge: Für jeden der acht primitven Typen PT bezeichnet der Plusoperator `+` *zwei* Konkatenationsoperationen:

```
String operator+(PT     pt, String st)
String operator+(String st, PT     pt)
```

Zu diesen 16 Konkatenationsoperationen kommen zwei Operationen mit einem `Object`-Operanden hinzu:

```
String operator+(Object ob, String st)
String operator+(String st, Object ob)
```

Die hier verwendete Syntax `operator+` zur Beschreibung von Operationen ("Funktionen mit einem Operator als Namen") wurde von C++ übernommen und darf in Java-Programmen *nicht* verwendet werden.

S. 141, Z. 13: Die Behauptung "[der zweite und dritte Operand des Bedingungsoperators `...?...:...`] dürfen zu *einem* beliebigen Typ (aber nicht zu zwei verschiedenen Typen) gehören" ist falsch. Die beiden Operanden dürfen zu verschiedenen Typen T1 und T2 gehören, wenn es von T1 nach T2 oder von T2 nach T1

eine *erweiternde Typumwandlung* gibt. Der Wert des gesamten Ausdrucks ist vom "weiteren der beiden Typen". Beispiele:

```
1  Ausdruck             T1      T2      Typ und Wert des Ausdrucks
2  true  ? 10.0 : 20   double  int     double 10.0
3  false ? 10.0 : 20   double  int     double 20.0
4
5  true  ? 'A'  : 20   char    int     int 65
6  false ? 'A'  : 20   char    int     int 20
```

Das (neue) Beispielprogramm `Operatoren08` enthält eine systematische Folge von Beispielen zu diesem Thema.

S. 163, in den nummerierten Zeilen 12 und 14 muss es statt `sr` an beiden Stellen `sbr` heißen (denn gemeint ist der formale Parameter `sbr`, der in Zeile 11 vereinbart wird).

S. 170, Z. -4: Statt "7.6 Reihungen vergleichen und sortieren" sollte es nur "7.6 Reihungen vergleichen" heißen, weil die Methode `Arrays.sort` zum Sortieren von Reihungen erst im folgenden Abschnitt 7.7 erwähnt wird. Schade, damit enthalten auch die automatisch erzeugten Seiten VII bis XII mindestens einen Fehler :-(.

S. 188, Z. 2: Statt `rfp.append("!");` sollte es `rfp.append("+");` heißen (sonst stimmen die weiter unten in den Zeilen 77 bis 86 wiedergegebenen Ausgaben nicht).

S. 213, Z. 7: Die Behauptung "Es kann aber durchaus vorkommen, dass der Modulaspekt einer Klasse leer ist." trifft nicht zu. Da zu jeder Klasse mindestens ein Konstruktor gehört und wir Konstruktoren zum Modulaspekt einer Klasse zählen, enthält der Modulaspekt einer Klasse mindestens *einen Konstruktor*.

S. 213, Z. -5: Wenn der Programmierer in einer Klasse keinen Konstruktor vereinbart, "schenkt" der Ausführer der Klasse einen Konstruktor mit 0 Parametern. Dieser Konstruktor hat *dieselbe Erreichbarkeit wie die Klasse* (d.h. er ist `public`, `protected`, paketweit erreichbar bzw. `private` je nachdem ob die Klasse `public`, `protected`, paketweit erreichbar oder `private` ist. Nur innere Klassen können `protected` oder `private` sein). Dank an Herrn Simon Effenberg für den Hinweis auf diesen Fehler.

S. 223, Z. 4: Diese Zeile ist mit 14 nummeriert. Statt "Wert des Ausdrucks" sollte es "Zielwert des Ausdrucks" heißen. Ziel des Autors war es, immer einfach und klar zwischen dem *Wert* und dem *Zielwert* einer Referenzvariablen (z. B. einer `String`-Variablen) zu unterscheiden. Hier ist ihm das misslungen.

S. 280, Z. 12: Diese Zeile ist mit 42 nummeriert. Statt
`} // druckeAnzahlPerson01` sollte es
`} // druckeAnzahlPerson02` heißen.

S. 289, Z. 19: Statt "(Nr. 11)" sollte es "(Nr. 13)" heißen.

S. 295, Z. 7 bis 12: Diese Zeilen sind mit 3 bis 8 nummeriert. Am Anfang jeder Zeile sollte es `tab` anstelle von `pr` heißen (z. B. `tab[0]` statt `pr[0]`, `tab[1]` statt `pr[1]` etc.).

S. 305, Z. -15: Die Cast-Regel-01 war bis Java 1.4 in Ordnung, ist in Java 5 aber unvollständig. Jetzt darf man per Cast-Befehl auch zwischen einem primitiven Typ und seiner Hüllklasse (in beiden Richtungen) umwandeln, z. B. von `int` nach `Integer` und von `Integer` nach `int`. Solche Cast-Befehle bewirken eine echte *Umwandlung*, nicht nur eine *Umdeutung*.

S. 351, Z. -14: Diese Zeile ist mit 34 nummeriert. Die hier skizzierte Methode `sortiere` ist kein gutes Beispiel, da sie als Parameter eine Reihung mit einem *rohen* Komponententyp (`Comparable`) erwartet. Weniger anstößig (aber auch etwas komplizierter) ist das folgende Beispiel:

```
1  static public
2     <K extends Comparable<K>> void sortiere1(K[] cr) {...}
```

Diese Methode `sortiere1` darf man auf alle Reihungen von Objekten anwenden, deren Komponenten miteinander *vergleichbar* sind (mit der Methode `comparTo` der Schnittstelle `Comparable`). In diesem Beispiel kommt kein roher Typ vor. Die Profi-Version der `sortiere`-Methode sieht noch etwas komplizierter aus:

```
1  static public
2     <K extends Comparable<? super K>> void sortiere2(K[] cr) {...}
```

Die Methode `sortiere2` kann man nicht nur auf Reihungen anwenden, deren Komponenten vom Typ K sind und sich mit anderen K-Objekten vergleichen lassen, sondern auch auf Reihungen, deren Komponenten vom Typ K sind und sich mit allen Objekten einer Oberklasse von K vergleichen lassen (z. B. sind in der Java-Standardbibliothek Objekte der Klasse `java.sql.Time` mit allen Objekten ihrer Oberklasse `java.util.Date` vergleichbar. Eine Reihung vom Typ `Time[]` könnte man mit `sortiere2` bearbeiten, aber nicht mit `sortiere1`).

S. 359, Z. -9: Statt "Die Beispielprogramme `Auswert01` bis `Auswert05` ..." sollte es heißen: "Die Beispielprogramme `Anmerk01` bis `Anmerk06` ...". Die neuen Namen sollen deutlicher an *Anmerkungen* (engl. annotations) erinnern. Vielen Dank an Sebastian Frodl, der mich auf den Fehler aufmerksam machte und die Korrektur vorschlug.

S. 387: Hier fehlt ein Hinweis auf eine relativ einfache Lösung des Problems: Man sollte in einem solchen Fall in einem `catch`-Block *geprüfte* Ausnahmen in *ungeprüfte* Ausnahmen "übersetzten", etwa so:

```
1  try {
2     ...
3     write(...); // Gefaehrlicher neuer Befehl, wirft evtl.
4                 // eine gepruefte IOException
5     ...
6  catch(IOException gepruefteAusnahme) {
7     RuntimeException ungepruefteAusnahme =
8        new RuntimeException(gepruefteAusnahme);
9     throw ungepruefteAusnahme;
10 } // try/catch
```

Vielen Dank an meinen Kollegen Prof. C. Knabe, der mich auf diese Lösung aufmerksam gemacht hat.

S. 400, Beispiel-01, nummerierte Zeilen 12 bis 21
Siehe unten S. 453, Beispiel-02

S. 427, Z. 14 (Anmerkung): Eine Klasse kann man normalerweise (d.h. wenn man sie direkt innerhalb eines Paketes vereinbart und nicht innerhalb einer umfassenden Klasse) nur mit dem Erreichbarkeitsmodifizierer `public` (als öffentliche Klasse) oder ohne diesen Modifizierer (als paketweit sichtbare Klasse) vereinbaren. Innerhalb einer umfassenden Klasse kann man eine Klasse als `public`, `protected`, paketweit sichtbares oder `private` Element vereinbaren.

S. 453, Beispiel-02: Eine `compareTo`-Methode ... `Mango`-Objekte
Diese `compareTo`-Methode liefert falsche Ergebnisse, wenn das `int`-Attribut `saftmenge` von `Mango`-Objekten auch extreme negative und positive Werte annehmen kann. Beispiel: Seien `ma` und `mb` zwei `Mango`-Objekte mit `ma.saftmenge` gleich -2 Milliarden, `mb.saftmenge` gleich +200 Millionen. Dann liefert der Aufruf `ma.compareTo(mb)` (wegen eines `int`-Überlaufs) einen positiven Wert (nämlich +2 094 967 296), obwohl `ma` *kleiner* ist als `mb`. Eine bessere `compareTo`-Methode, die `Mango`-Objekte nach ihrer `int-saftmenge` vergleicht, kann etwa so aussehen:

```
1     public int compareTo(Mango that) {
2        if (this.saftmenge < that.saftmenge) return -1;
3        if (this.saftmenge > that.saftmenge) return +1;
4        return 0;
5     }
```

S. 459, Z. -4: Der Hinweis "funktionierte aber bei der Java-Version 5.0 offenbar noch nicht ganz richtig" ist falsch. Die Methode `Collections.checkedCollection` funktionierte richtig, aber das Beispielprogramm `Sammlungen07` war

grundsätzlich falsch konzipiert. Inzwischen ist es verbessert und demonstriert jetzt die Wirkung der Methode `Collections.checkedCollection`.

S. 510, Z. 1: Die Definition "Eine Methode ist *fadensicher*, wenn sie auch bei Ausführung durch mehrere Fäden richtig funktioniert" ist unglücklich formuliert (d. h. falsch). Viele Methoden werden dadurch fadensicher gemacht dass man (mit dem `synchronized`-Befehl) eine gleichzeitige Ausführung durch mehrere Fäden *unmöglich* macht. Auf diese fadensicheren Methoden trifft obige Definition aber nicht wirklich zu.

Es folgt eine (hoffentlich) bessere Definition: "Eine Methode, die auch dann richtig funktioniert, wenn mehrere Fäden versuchen, sie nebenläufig auszuführen, bezeichnet man als *fadensicher* (engl. thread safe)".

S. 540, Z. -9: Statt "der Art `windowClosing`" sollte es "der Art `actionPerformed`" heißen (weil auf S. 539 in der nummerierten Zeil 108 `actionPerformed` steht, und nicht `windowClosing`).

S. 543, Z. 5: Statt "zwei oberste Grabo-Pakete: `awt` ..." sollte es besser "zwei oberste Grabo-Pakete: `java.awt` ..." heißen (weil es danach auch `javax.swing` und nicht nur `swing` heißt).

S. 550, Z. -3: Statt "Beispielprogramm `Grabo08Tst`" sollte es "Beispielprogramm `Katze`" heißen. Das Programm heißt jetzt `Katze`, weil es Mausereignisse fängt. Ein Beispielprogramm `Grabo08Tst` gibt es nicht mehr.