



Unbe-rechnen-bar

Sicherer Umgang mit Zahlen und Daten

Holger Schulz

Warum ist in Java $3 * 0.5$ gleich 1.5 , aber $3 * 0.4$ nicht 1.2 ? Wieso bekommt man auf einem Mac andere Rechenergebnisse als unter Windows und warum speichert die Datenbank die Zahl -2 als 254 ab? Wer glaubt, dass Computer gut rechnen können und Datenbanken Daten sichern, wird in diesem Artikel vom Gegenteil überzeugt.

Die Verarbeitung von Zahlen und das Verhalten von Datenbanken bzw. Treibern kann – wenn man Glück hat – zu offensichtlichem Fehlverhalten eines Programms führen. Schlimmer sind jedoch die unauffälligen, kleinen Rundungsfehler bei der Währungsumrechnung und der totale Datenverlust beim Sichern in die Datenbank. Anhand von leicht nachvollziehbaren Beispielen vermittelt dieser Artikel einen Einblick, wie Java Zahlen verarbeitet und welche Konsequenzen dies in der Praxis hat.

Java ist beschränkt

Um genau zu sein: auf 64 Bits. Damit können beim Datentyp `long` 2^{64} unterschiedliche ganze Zahlen gespeichert werden. Dem Datentyp `double` steht dieselbe Anzahl Bits zur Verfügung, weshalb auch dieser nicht mehr verschiedene Werte speichern kann. Für `integer` und `float` gilt dies analog mit 32 Bits.

Dieser einfache Sachverhalt hat zum Beispiel zur Konsequenz, dass der Euro-Umrechnungskurs von **1.95583** sich nicht exakt als `double` oder `float` abbilden lässt. `Float` kann keine Zahl zwischen **1.9558299779891967734375** und **1.955830097198486328125** speichern. `Double` stehen zwar mehr Bits zur Verfügung – weshalb die Zahlen dichter an **1.95583** liegen –, gerechnet wird jedoch auch hier mit Näherungswerten.

Irrglaube

Die Ausgabe von `System.out.println(1.95583f)` ist **1.95583** und verleitet zu dem Trugschluss, dass Java mit dieser Zahl rechnen würde. Die Ausgabe weicht jedoch fast immer vom eigentlichen Wert ab. Näheres erläutert die Beschreibung von `Float.toString()`:

“[...]There must be at least one digit to represent the fractional part, and beyond that as many, but only as many, more digits as are needed to uniquely distinguish the argument value from adjacent values of type float. [...]”

Konsequenzen

Auch wenn Java für `float` und `double` die Zahl **1.95583** ausgibt, wird intern doch mit den Näherungswerten gerechnet. Da diese für `float` und `double` unterschiedlich sind, gilt $1.95583f \neq 1.95583d$.

Diese Tatsache beantwortet auch die eingangs gestellte Frage, warum $3 * 0.5$ gleich 1.5 , aber $3 * 0.4$ nicht 1.2 ist. Die Zahlen **3**, **0.5** und **1.5** lassen sich exakt als `double` speichern. Die Rechnung $3 * 0.4$ ergibt jedoch einen internen Wert knapp oberhalb von **1.2**, während die Zahl **1.2** intern als **1.19999...** gespeichert wird.

Mit Strings rechnen

Genauer als der `float`-Wert **1.95583f** ist der String `"1.95583"`. So liefert die Ausgabe `System.out.println(new BigDecimal("1.2"))` auch



exakt **1.2** während `System.out.println(new BigDecimal(1.2))` den Wert **1.19999...** ausgibt. Im Gegensatz zu `float` und `double` darf man der Ausgabe von `BigDecimal` trauen. Berechnungen, die mit `BigDecimal("1.2")` durchgeführt werden, erfolgen tatsächlich mit dem Wert **1.2**. Die Rechengenauigkeit erkauft man sich jedoch mit einer wesentlich langsameren und umständlicheren Berechnung.

Merke: *„floating-point is designed to give you the wrong answer very quickly.“*

Hierzu einige Beispiele:

```
(A) System.out.println(10000.0f * 1.95583f);
    // 19558.299
(B) System.out.println(new BigDecimal(10000.0f * 1.95583f));
    // 19558.298828125
(C) System.out.println(10000.0d * 1.95583d);
    // 19558.3
(D) System.out.println(new BigDecimal(10000.0d * 1.95583d));
    // 19558.299999999997240423858165740966796875
(E) System.out.println((new BigDecimal(10000)).
    multiply(new BigDecimal(1.95583)));
    // 19558.2999999999995743849012796999886631965637207031250000
(F) System.out.println((new BigDecimal("10000")).
    multiply(new BigDecimal("1.95583")));
    // 19558.30000
```

An diesem Beispiel wird deutlich, wie sich Berechnungen mit `float`, `double` und `BigDecimal` auswirken:

- Die Ausgabe der Berechnung mit `float` unterscheidet sich vom exakten Wert **19558.3**.
- Der „interne Wert“ – mit dem gerechnet wird – unterscheidet sich von dem in (A) ausgegebenen Wert.
- Die Berechnung mit `double` scheint korrekt.
- Hier wird der „interne Wert“ der `double`-Berechnung aus (C) ausgegeben.
- In (D) wurde das Rechenergebnis wieder auf einen `double`-Wert gerundet – (E) gibt das korrekte Ergebnis der Multiplikation der beiden `double`-Werte aus.
- Der einzig korrekte Weg zur Multiplikation zweier „Zahlen“.

Konsequenzen

Wer mit Näherungswerten und Rundungsfehlern nicht leben kann, muss mit `Strings` und `BigDecimal` rechnen. Was aber, wenn die zugrundeliegenden Zahlen nur als `double` vorliegen?

Meist möchte man mit dem ausgegebenen – nicht mit dem internen – Wert rechnen. Eine Brücke zur exakten Berechnung erhält man mit folgendem Hilfskonstrukt:

```
double dEuro = 1.95583;
BigDecimal bdEuro = new BigDecimal(Double.toString(dEuro));
```

Man muss sich jedoch bewusst sein, dass auch die Ausgabe von `double` nicht immer dem gewollten Wert entspricht.

Runden

Dass 7.5 aufgerundet und 7.4 abgerundet wird, lernt man in der Schule. Doch schon bei -7.5 scheiden sich die Geister: -7 oder -8?

Die Klasse `BigDecimal` verlangt für manche Rechnungen explizit die Angabe, wie gerundet werden soll – wobei acht Verfahren zur Auswahl stehen. Für eine Bank macht es einen erheblichen Unterschied, ob täglich eine Millionen halbe Cent auf- oder abgerundet werden. Zu einer exakten Berechnung gehört somit auch die Angabe, wie gerundet werden soll.

Java kennt -0.0

Neben den Besonderheiten wie `Infinity` und `NaN` kennen `float` und `double` auch die Zahl `-0.0`. So liefert die Rechnung `-3.0*0.0` die Ausgabe `-0.0`. Wer sich in Sicherheit wiegt, weil für die primären Datentypen `0.0==0.0` gilt, tappt in die Bitfalle: Die Klassen `Float` und `Double` vergleichen Zahlen nach den gesetzten Bits. Um `0.0` von `-0.0` unterscheiden zu können, belegen die Zahlen unterschiedliche Bits und folglich gilt auch nicht `new Double(0.0).equals(new Double(-0.0))`.

Mythos strictfp

`strictfp` dürfte eines der am seltensten genutzten Schlüsselwörter in Java sein. Und wenn es eingesetzt wird, dann oft in der Hoffnung, die Rechengenauigkeit zu erhöhen. Dazu ein (Gegen-)Beispiel. Die Klasse `DemoStrictfp` enthält folgende Methoden:

```
public strictfp static double getStrict(double _d) {...}
public static double getNormal(double _d) {...}
```

Beide Methoden enthalten exakt den gleichen Code und führen dieselben Rechenoperationen mit `double`-Werten durch. Während die Methode `getStrict` auf allen Rechnern exakt gleiche Ergebnisse zurückliefert, ist das Ergebnis von `getNormal` prozessorabhängig.

Konsequenzen

Wenn man sicherstellen muss, dass ein Programm rechnerunabhängig dieselben Rechenergebnisse liefert, müssen entsprechende Methoden oder Klassen explizit mit `strictfp` gekennzeichnet werden. `strictfp` erzwingt, dass *floating point*-Berechnungen *strikt* nach der IEEE-Norm durchgeführt werden.

Ohne `strictfp` erlaubt Java, Zwischenberechnungen abweichend von der IEEE-Norm durchzuführen, damit die wesentlich schnelleren, prozessorabhängigen Berechnungsalgorithmen genutzt werden können. Aus diesem Grund gibt es in Java auch zwei Klassen für die gleichen mathematischen Berechnungen: `Math` und `StrictMath`. Aus der Beschreibung zur Klasse `java.lang.Math`:

„[...]Unlike some of the numeric methods of class `StrictMath`, all implementations of the equivalent functions of class `Math` are not defined to return the bit-for-bit same results. This relaxation permits better-performing implementations where strict reproducibility is not required.[...]“

	Standard	mit <code>strictfp</code>	mit <code>BigDecimal</code>
genaue Berechnung	—	—	✓
rechnerunabhängig	—	✓	✓
einfach anzuwenden	✓	✓	—
performant	✓	(✓)	—

Tabelle 1: `strictfp` versus `BigDecimal`

strictfp vs. BigDecimal

Mit `BigDecimal` werden Berechnungen genau durchgeführt – mit `strictfp` auf allen Rechnern lediglich gleich (ungenau). Tabelle 1 gibt eine Übersicht.

Cast-Falle Wertebereich

Als *Cast* bezeichnet man die Umwandlung von einem Datentyp in einen anderen. Während die Umwandlung von `byte`- in `integer`-Werte gefahrlos ist, kann der umgekehrte Weg – von einem großen in einen kleinen Wertebereich – zu einem totalen Datenverlust führen. Deshalb verlangt Java in solchen Fällen auch die explizite Angabe des *Cast*-Operators. Beispiel:

```
int n0 = 77;
byte b0 = (byte)n0;
```

Dieses Beispiel verursacht keine Probleme, da Java für `byte` den Wertebereich `-127 bis 128` definiert hat. Weist man `n0` jedoch den Wert `200` zu, so erhält `b0` den Wert `-56`! Java verlässt sich beim Casten auf den Programmierer und führt keinerlei Überprüfungen durch. Jeder Wert außerhalb dieses Bereichs wird in eine völlig andere Zahl gecastet.

Konsequenzen

Diese Tatsache ist allgemein bekannt und doch tappt man leicht in diese Falle – nicht im eigenen Code, sondern zum Beispiel bei der Verwendung von Datenbanken beziehungsweise `JDBC`-Treibern.

Einige Datenbanken – zum Beispiel `MySQL` – definieren in der Defaulteinstellung den Wertebereich für `bytes` wie in Java von `-127 bis 128`. Andere – zum Beispiel `Access` – erlauben Werte von `0 bis 255`.

Ist in der `Access`-Datenbank ein `byte`-Feld mit dem Wert `255` belegt und wird dieser mit `resultSet.getBytes(...)` ausgelesen, so erhält man ohne Warnung* den Wert `-1`. Möchte man umgekehrt einen Wert `-2` in ein `byte`-Feld schreiben, so wird jedoch der Wert `254` gespeichert.

Einige `JDBC`-Treiber verhindern das Schreiben unzulässiger Werte, die meisten Treiber verlassen sich jedoch – unter anderem aus Performancegründen – auf den Programmierer.

Für das sichere Schreiben in Datenbanken sollten eigene Prüf- und Konvertierungsroutinen vorgeschaltet werden. Ansonsten kann bei der Migration von Daten aus einem Datenbanksystem in ein anderes der totale Datenverlust drohen, ohne dass man dies sofort bemerkt.

Cast-Falle Rundung

Während man bei Java sicher sein kann, dass unterschiedliche „innere Werte“ zu unterschiedlichen Ausgaben der Zahlen führen, ist dies bei Datenbanken (hier `Access`) nicht unbedingt der Fall.

*`JDBC`-`ODBC`-Bridge von Sun in `JDK 1.5.0_04-b05`



afloat	
	0,1
	0,1

Abb. 1: Anzeige von Gleitkommazahlen in Access

Expr1000	
	100,000008940697
	100,000001490116

Abb. 2: Abfrage in Access

In der ersten Zeile von Abbildung 1 wurde der Wert `0,10000001` in der zweiten Zeile `0,1` eingegeben. Access zeigt beide Werte stets als `0,1` an.

Die Abfrage `SELECT afloat*1000 FROM FLOATTEST;` (s. Abb. 2) zeigt, dass die Daten – trotz gleicher Anzeige – unterschiedlich sind.

Konsequenzen

In einem Java-Programm ausgelesen, werden die Zahlen `0.10000001` und `0.1` ausgegeben. Java verhält sich hier völlig korrekt. Dass Access – als Oberfläche für die zugrundeliegende Datenbank – die Daten nicht korrekt anzeigt, ist Java nicht anzulasten. Es verdeutlicht jedoch das Problem, dass Rundungen an unterschiedlichen Stellen

erfolgen können und auch bei Datenbanken nicht immer mit den Zahlen gearbeitet wird, die angezeigt werden.

Datum und Datenbanken

Ein Datum ist im Grunde nichts anderes als eine Zahl, die in Java als Anzahl der Millisekunden seit 1970 repräsentiert wird. Die Millisekunden sind dabei auf UTC (Coordinated Universal Time) bezogen – enthalten somit weder Verschiebungen durch Sommerzeit noch durch Zeitzonen. Diese Angabe wäre deshalb ideal, um in einer international genutzten Datenbank gespeichert zu werden. Jeder könnte in seinem Land – bei der Abfrage der Daten aus der Datenbank – den Zeitpunkt in seine lokale Uhrzeit umgerechnet bekommen.

Leider sieht die Praxis anders aus, und beim Auslesen dieses Datums erhält man überall auf der Welt dieselbe Uhrzeit angezeigt.

Möchte man das aktuelle Datum zum Beispiel in Access speichern, ist folgende Zeile naheliegend:

```
preparedStatement.setObject(1, new Date());
```

Die JDBC-ODBC-Bridge von Sun wird die Ausführung jedoch mit einer SQL-Exception „Unknown SQL Type“ ablehnen. Dieser JDBC-Treiber benötigt explizit ein Timestamp-Object:

```
Date date = new Date();
preparedStatement.setObject(1, new Timestamp(date.getTime()));
```

Konsequenzen

Es ist somit eine treiberabhängige Konvertierung notwendig, bei der in Java zusätzliche Daten (Nanosekunden) hinzugefügt werden. Dass Access nicht einmal Millisekunden speichern kann, führt diese Konvertierung ad absurdum. Andere Datenbanken würden die Nanosekunden speichern und somit eine Genauigkeit vortäuschen, die die Ursprungsdaten nicht besaßen.

Beim Wechsel eines Datenbanksystems ist somit zu prüfen, ob die Daten unverändert übernommen werden können. Prüf-

und Transformationsroutinen sollten dem JDBC-Treiber vorgeschaltet werden, um ggf. vor Datenverlust zu warnen und die Ursprungsdaten treibergerecht zu konvertieren.

Datenfalle JDBC

Einen JDBC-Treiber kann man sich am besten als eine Black-Box vorstellen, die man mit Daten füttert und die nach „geheimen“ Transformationen Zahlen in eine Datenbank schreibt. Wie weiter oben beschrieben, können die gespeicherten Zahlen völlig von den Eingangsdaten abweichen, wenn diese nicht zum Datentyp der Datenbank passen. Dabei kann das Verhalten groteske Züge annehmen:

Hat man in Access eine Spalte vom Typ `decimal(20,17)` angelegt und speichert nun die Zahl `BigDecimal("0.3")`, wird korrekt `0.3` gespeichert. Dagegen wird `BigDecimal(0.3)` – ohne Anführungsstriche – nach den „geheimen“ Transformationen der JDBC-ODBC-Bridge in der Datenbank als Zahl `0` gespeichert!

`BigDecimal(0.3)` entspricht dem Wert `0.2999999999999999888...` mit 54 Nachkommastellen. Für die JDBC-ODBC-Bridge muss der Scale zuvor auf die erlaubten 17 Stellen reduziert werden:

```
double d = 0.3;
BigDecimal bd = new BigDecimal(d);
bd = bd.setScale(17, BigDecimal.ROUND_HALF_UP);
preparedStatement.setBigDecimal(1, bd);
```

Nach dieser Vorbereitung wird in der Datenbank der Wert `0.2999999999999999999` gespeichert. Das ist schlüssig, denn dies ist der auf 17 Nachkommastellen gerundete Wert von 0.3. Der Lösungsweg zum Speichern von 0.3 lautet:

```
double d = 0.3;
BigDecimal bd = new BigDecimal(Double.toString(d));
```

Konsequenzen

Neben den Datenbanken besitzen auch (fast) alle JDBC-Treiber ihre Eigenheiten. Der Umgang mit Datenbanken und Treibern erfordert wesentlich mehr Sorgfalt und Misstrauen als der Name „Datenbank“ vermuten lässt.

Schlusswort

Wer jetzt Java den Rücken kehren will, wird erneut enttäuscht werden. Den physikalischen Grenzen unserer Digitalrechner erliegen alle Programmiersprachen und Datenbanken. Java bietet jedoch – wie gezeigt – Sprachkonstrukte und Klassen, um der Physik Herr zu werden.



Dipl.-Ing. Holger Schulz ist Entwicklungsleiter für ETL/EDI- und Datenbankssoftware.
E-Mail: holger.schulz@datenschupser.de.



Weitere Informationsquellen

<http://www.javaranch.com/newsletter/July2003/newsletterjuly2003.jsp#a4>