Oktober 2012

Javas Date ist kein echtes Datum

Dr. Jürgen Lampe
Agon Solutions

Abstract

Java fehlt es bis heute an einem durchdachten Konzept für die Abbildung des Datums. Der ohne gründliche Überlegungen eingeführte Typ `java.util.Date` ist nichts anderes als ein unreflektiert aus der UNIX-Welt übernommener Zeitstempel. In der Praxis führt das immer wieder zu unerwarteten Fehlern oder unnötig aufwendigen Lösungen, weil bei der Zuordnung eines Zeitpunkts zu einem Datum im Detail viele Fallen lauern.

In vielen, insbesondere geschäftlichen Anwendungen ist das Datum ohne Zeit ausreichend, wenn es beispielsweise um Löschfristen oder Tagegeldberechnungen geht. Ausgehend von einer Analyse der Probleme, die sich in solchen Fällen aus der Verwendung des Java-Dates ergeben, wird eine einfach zu implementierende und performante Alternative vorgestellt.

Die Klasse java.util.Date

Dass diese Java-Klasse nicht besonders gut gelungen war, ist wohl auch den Entwicklern bei SUN recht schnell bewusst geworden. Inzwischen sind 22 der 33 öffentlichen (public) Methoden bzw. Konstruktoren als überholt (deprecated) gekennzeichnet. Allerdings ist eine echte Aufarbeitung bisher leider unterblieben, was heißt, dass innerhalb der Java-Bibliotheken diese problematischen Funktionen weiterhin verwendet werden. So verwendet z. B. die statische Methode `java.sql.Date.valueOf(String)` immer noch den überholten Konstruktor `Date(int, int, int)`. Das kann zu unerwarteten Ergebnissen führen, wenn sich die Zeitzoneneinstellung der Datenbank von denen der Java-Anwendung unterscheidet.

Bereits mit Version 1.1 wurden die Klassen `java.util.Calendar/GregorianCalendar` kreiert, um die offensichtlichsten Defizite beim Umgang mit Date zu beheben. Grundsätzlich leisten diese Klassen bei konsequenter Anwendung das Erwartete. Allerdings gibt es zwei problematische Punkte:

- Kalender benötigen immer eine Zeitzone. Für die bequemere Verwendung muss diese jedoch nicht unbedingt explizit angegeben werden. Falls sie fehlt, wird die Default-Zone der jeweiligen Installation genommen. Da das gängige Programmierpraxis ist, erhält man implizite Abhängigkeiten von der Installation.
- Die Berechnungen des Kalenders sind relativ kompliziert und die Erzeugung eines Kalenderobjekts ist aufwendig. Wenn Datumskonvertierungen oder -berechnungen sehr häufig benötigt werden, sind spezielle Maßnahmen erforderlich, um zu vermeiden, dass daraus ein Leistungsengpass entsteht.

Die diffuse Interpretation des Datum-Begriffs findet sich auch in der JDBC-Spezifikation wieder. Da `java.sql.Date` eine Subklasse von `java.util.Date` ist, hat man es de facto immer mit einem Zeitpunkt zu tun, der erst in ein Datum konvertiert werden muss. Das kann

zu subtilen Fehlern führen, wenn man sich auf die Standardeinstellungen verlässt, weil SQL für das DATE keine Zeitzone kennt (nur für TIME und TIMESTAMP).

Die erforderliche explizite Behandlung durch z. B. Aufruf der `getDate`–Methode mit Calendar-Parameter (seit JDBC 2.0) ist eine völlig unangemessene Verkomplizierung, die überdies Leistung kostet. Überdies kann die im Allgemeinen unterschiedliche und nicht immer standardkonforme Implementierung des DATE-Typs von SQL in den verbreiteten Datenbanken zu weiteren subtilen Problemen führen[1].

Ein ganz spezielles Problem kann beim Remote Procedure Call (RPC) oder anderen Protokollen, die Objektserialisierung verwenden, auftreten, wenn die beteiligten Seiten mit unterschiedlichen Zeitzonen-Einstellungen arbeiten. Weil Datumswerte in der Regel als Zeitstempel Mitternacht der entsprechenden Zeitzone realisiert werden, führt das bei der Übertragung ohne spezielle Vorkehrungen in einer Richtung immer zu einer Verschiebung um einen Tag. (In der anderen Richtung nur zu einer dem Zeitzonen-Offset entsprechenden Stundenverschiebung.) Dafür kann es keine generelle Lösung geben, weil dieses Verhalten für Zeitstempel absolut korrekt ist und die Serialisierung nicht unterscheiden kann, ob ein Date-Objekt nun einen Zeitpunkt oder ein reines Datum darstellt.

Aus heutiger Sicht war es ebenfalls falsch, Date-Objekte nicht unveränderbar zu machen, ähnlich `Integer` oder `Long`. [1] Das ist zwar keine prinzipielle Frage, aber die Veränderbarkeit hat an vielen Stellen für Verwirrung gesorgt.

Das Datum

Das Datum ist eines jener scheinbar trivialen Konzepte, die jeder zu verstehen meint, die bei genauer Betrachtung aber ihre Tücken haben. Beim Datum erkennt man das schnell, wenn man an die unterschiedlichen Tage denkt, an denen katholische und orthodoxe Christen Weihnachten feiern.

Es gibt weitere Kalender, denen allen gemeinsam ist, dass sie auf Tagen als Grundeinheit basieren. Die Probleme mit Javas Date beruhen nicht zuletzt darauf, dass der Tag aus einer feingranulareren Zeitskala abgeleitet wird. In vielen Fällen, vor allem bei technischen Prozessen, sind Tage tatsächlich ein zu grober Maßstab. Gerade bei wirtschaftlichen Prozessen gibt es jedoch zahlreiche Vorgänge, die auf der Basis von Tagen operieren. Bei Zinstagen zeigt sich das schon im Namen. Ein Lieferdatum kann beim Lieferanten unterschiedliche Zeitpunkte beschreiben, wenn das Datum des Erfüllungsorts entscheidend ist. Andere Beispiele sind tagesbezogene Mieten, Belegtage von Betten oder

[1] Bei Oracle, wo der Typ DATE immer mit Zeit, aber standardmäßig ohne Zeitzone abgelegt wird (*If no time was specified when the date was created, the time defaults to midnight* [4]) findet sich folgender Hinweis: *There is little need to use the Oracle external DATE datatype in ordinary database operations. It is much more convenient to convert DATE into character format, because the program usually deals with data in a character format, such as DD-MON-YY.* (a. a. O.)

Besucherzahlen. All diesen Fällen ist gemeinsam, dass solche Details wie Schaltsekunden oder Zeitzonenwechsel unerheblich sind. Niemand verlangt für ein über das letzte März- oder Oktoberwochenende geliehenes Auto wegen des Zeitzonen-Wechsels – was die Sommer-/Winterzeitumstellung letztlich ist – ein höheres oder niedrigeres Entgelt.

In anderen Programmiersprachen bzw. –systemen wird dem Rechnung getragen. COBOL hat eine Funktion (CEECBLDY), die ein gültiges Datum in einen Integer-Wert umrechnet, der die Zahl der Tage seit dem 31.12.1600 angibt.

Microsoft Excel und einige andere Programme stellen Datumswerte (mit geringfügigen Abweichungen) als Zahl der Tage seit dem 1.1.1900 dar, wobei 1900 von Excel fälschlicherweise als Schaltjahr behandelt wird. Excel für Mac beginnt die Zählung erst 1904, wodurch der Schaltjahrfehler einfach umgangen wird. Vergleichbare Hilfsmittel fehlen in Java.

Eine echte Datumsklasse in Java

Eigentlich wäre es längst an der Zeit gewesen, dass Java um Features zum effizienten Umgang mit reinen Datumswerten erweitert wird. Da das aber nicht passiert ist und wohl so schnell auch nicht passieren wird, sollen hier ein paar Vorschläge und Hilfen präsentiert werden, die es ermöglichen, die gewünschte Funktionalität problemlos selbst zu implementieren.

Wichtig ist es, von Anfang an, d. h. bereits im Design, klar zu unterscheiden, wo ein reines Datum ausreicht und wo doch auf einen Zeitstempel (Date) zurückgegriffen werden muss. Es ist nicht immer ganz offensichtlich, ob ein Datum plus Uhrzeit oder ein Date die passende Wahl ist. Beispielsweise muss man dazu unterscheiden, ob etwas täglich zur gleichen Uhrzeit oder alle 24 Stunden ausgeführt werden soll.

`(java.util.Timer.schedule(TimerTask task, Date firstTime, long period)` macht das Letztere.).

Was im allgemeinen Verständnis zunächst nach eine spitzfindigen Unterscheidung aussieht, hat bei jeder Zeitumstellung Konsequenzen.

Auf jeden Fall vermeiden sollte man die wiederholte Konvertierung von Date nach Datum und zurück. Das verbraucht nicht nur unnötige Leistung, sondern ist ein Indiz dafür, dass der Entwurf nicht konsistent ist. Es kann natürlich sein, dass eine projektexterne Schnittstelle, der man sinnvollerweise ein Datum übergeben sollte, ein Date erfordert. Die beste Lösung, die Schnittstelle zu modifizieren, wird nicht immer möglich sein. Dann ist es eine Frage der Abwägung, ob die Vorteile in anderen Teilen der Anwendung den zusätzlichen Aufwand an der Schnittstelle rechtfertigen.

Wenn Datumswerte in eine Datenbank geschrieben oder aus dieser gelesen werden sollen, muss unbedingt geprüft werden, wie das so erfolgen kann, dass die gewonnenen Vorteile dabei nicht wieder verloren gehen. Insbesondere sollte man Konvertierungen in und aus Date unbedingt vermeiden. Die einfachste Lösung ist dabei das Ablegen als Ganzzahl. Sie hat aber den Nachteil, dass die Interpretation bei direkten SQL-Abfragen umständlicher ist. Möglicherweise bietet die Datenbank dafür geeignete Hilfsfunktionen oder man kann entsprechende Stored Procedures selbst schreiben. Diese Form hat Vorteile, wenn auch auf der Datenbank Berechnungen auf Tagesbasis ausgeführt werden sollen. Als Alternative bietet sich ein Abspeichern als String (z. B. der Form YYYY-MM-DD) an. Das erfordert zwar vor Berechnungen jeweils eine Konvertierung, ist aber für den Menschen leichter lesbar[2].

Eine weitere Frage, die möglichst früh geklärt werden muss, ist die nach dem benötigten Bereich gültiger Datumswerte. Ein erheblicher Teil des Aufwands bei Javas Kalenderberechnungen wird durch die Berücksichtigung von Datumsumstellungen (Julianisch nach Gregorianisch) und Zeitzonenveränderungen verursacht. Die Zeitzonen sind für das reine Datum außer bei Konvertierungen von und nach Date irrelevant. Für den weit überwiegenden Teil aller Projekte ist die Begrenzung auf die Gregorianische Zeitrechnung keine Einschränkung. Anwendungsgebiete, in denen das nicht ausreichend ist, werden mit den Standardmitteln ohnehin nicht zurechtkommen, da es beispielsweise in Deutschland keinen einheitlichen Kalenderwechsel gab und in einigen Gebieten beide Kalender über Jahrzehnte parallel verwendet wurden. Solche Anwendungen werden hier nicht betrachtet. Wir beschränken uns auf den Gregorianischen Kalender und Daten vom 1.1.1583 bis zum 31.12.9999.

Die Julianische Tageszahl

Prinzipiell kann man natürlich jedes beliebige Datum auswählen, um davon startend die Tage zu zählen. Es erleichtert aber die Kommunikation und das Verständnis, wenn man auf eine Konvention setzt.

In diesem Beitrag wird das chronologische Julianische Datum als Berechnungsbasis verwendet. Der Name ist etwas irreführend, da es keinen unmittelbaren Bezug zum Julianischen Kalender gibt. [2]

Vorteil dieser Wahl ist, dass die benötigten Algorithmen erprobt und aufbereitet verfügbar sind. Überdies gibt es Online-Tools zur Berechnung, z. B. [3], die den Test erleichtern. Die in [2] angegebenen Umrechnungsalgorithmen sind stabil gegenüber Wertüberschreitungen, so dass man beispielsweise für das Zahlentripel (0, 8, 2012) die gleiche Julianische Tageszahl erhält wie für (31, 7, 2012). Das gilt in analoger Weise für den Monatswert und ist nicht auf die 0 beschränkt, es sind beliebige positive und negative Werte zulässig, solange man den definierten Datumsbereich dadurch nicht verlässt. Das Datum des Ablaufs einer Monats-

[2] In [5] wird vorgeschlagen, Zeitpunkte grundsätzlich als UTC-Millisekundenwert in der Datenbank zu speichern, um u. A. die Tatsache, dass bei der Konvertierung immer eine Zeitzone notwendig ist, explizit zu machen.

oder Tagesfrist kann damit sehr einfach berechnet werden, indem der entsprechende Ausgangswert einfach um die Frist vergrößert wird und anschließend ein Hin- und Rückkonvertierung ausgeführt wird.

Listing 1

```
public class Day {

    final int year, month, day;

    String asString;

    public Day(int year, int month, int day) {
        this.year= year;
        this.month= month;
        this.day= day;
    }

    public int getYear() {
        return year;
    }

    public int getMonth() {
        return month;
    }

    public int getDay() {
        return day;
    }

    @Override
    public int hashCode() {
        return year * 400 + month * 31 + day;
    }

    @Override
    public boolean equals(Object obj) {
        boolean result= false;
        if (obj instanceof Day) {
            Day dayobj= (Day) obj;
            result= dayobj.year==year && dayobj.month==month
                    && dayobj.day==day;
        }
        return result;
    }

    @Override
    public String toString() {
        if (asString==null){
            asString= year + "-" + (month<10?"0":"") + month +
                      "-" + (day<10?"0":"") + day;
        }
        return asString;
    }
}
```

Desweiteren kann auf einfache Weise mittels Modulo 7 der Wochentag berechnet werden. Der Wert null entspricht dabei dem Montag, eins dem Dienstag usw. bis sechs dem Sonntag.

Implementierung

Es folgt nun die exemplarische Implementierung der ausgewählten Strategie. Wir beginnen mit einer Containerklasse für das Datum (Listing 1). Kommentare wurden der Übersichtlichkeit halber weggelassen. Diese Klasse enthält als reiner Container keine Logik, die die oben erwähnte Stabilität gegenüber scheinbar unsinnigen Werten berücksichtigt, d. h.

```
new Day(2012, 8, 0).equals(new Day(2012, 7, 31))
```

ist immer falsch.

Listing 2

```
public class JulianDayConverter {

    public static int calculateJD(int year, int month, int day) {
        int y= 240 * year + 20 * month - 57;
        int a= ((367 * y / 240) * 4 - 7 * (y / 240)  + 4 * day) / 4;
        int b =  (4 * a - 3 * (y / 24000)) / 4;
        int jd = b + 1721115;
       return jd;
    }

    public static int calculateJD(Day gregorianDay) {
        return calculateJD(gregorianDay.getYear(),
                gregorianDay.getMonth(), gregorianDay.getDay());
    }

    public static Day createDayFromJD(int julianDayNumber){
        int g= ((julianDayNumber << 2) - 7468865) / 146097;
        int a= julianDayNumber + 1 + g - (g >> 2);
        int b= a + 1524;
        int c= (20 * b - 2442) / 7305;
        int d= (1461 * c) >> 2;
        int e= 10000 * (b - d) / 306001;
        int day= b - d - 306001 * e / 10000;
        int month= e < 14 ? e - 1 : e - 13;
        int year= month > 2 ? c - 4716 : c - 4715;
        return new Day(year, month, day);
    }
}
```

Die Umrechnung zwischen Day bzw. den Integer-Werten für Jahr, Monat und Tag und der Julianischen Tageszahl (JD) erfolgt durch statische Methoden der Klasse JulianDayConverter (Listing 2). Die Algorithmen wurden aus [2] übernommen und lediglich so umgeformt, dass sie mit Integer-Arithmetik auskommen. Für ihre Herleitung wird auf die dort angegebene Literatur verweisen. Der Code enthält noch keine Überprüfungen, ob die Beschränkung auf das Zeitintervall 1.1.1583 bis 31.12.9999 eingehalten wird. Es hängt vom konkreten Projekt ab, ob ein solcher Test wirklich erforderlich ist.

Bei Bedarf kann die Umrechnung auf Julianische Daten (d. h. Datumsangaben des Julianischen Kalenders) oder andere Kalender erweitert werden. Außerdem kann es sinnvoll

sein, Versionen der Methode `createDayFromJD` zu haben, die jeweils nur einen der Werte Tag, Monat oder Jahr zurückliefern, wenn diese häufig gebraucht werden, um so unnötige Instanziierungen von Day-Objekten zu vermeiden.

Vorteile

Im Anhang finden sich einige Beispiele für die Verwendung der Methoden. Die Vorteile der Julianischen Tageszahl sind klar ersichtlich:

- Fortlaufende Folge, deren Teilfolgen leicht zur Indexierung von Feldern und Listen benutzt werden können
- Einfache Berechnung von Fristen oder Tageszahlen
- Effiziente Umwandlung von und nach (Tag, Monat, Jahr)[3]
- Einfache String-Konvertierung

Die Umrechnung erfolgt durch statische Methoden, die auch intern keinerlei Objekte anlegen. Das ist ein wichtiger Unterschied zur Verwendung von Javas Calendar-Klasse, die aufwendig initialisiert werden muss und die überdies nicht thread-safe ist.

Dadurch, dass hier der Umweg über einen Zeitpunkt – wie er implizit bei allen Datumsrechnungen in Java gegangen wird – vermieden wird, vereinfacht sich die Konvertierung aus dem und in das String-Format ganz entscheidend. Es bleiben nur drei Integer-Werte umzuwandeln, ohne jede weitere Berechnung.

Konvertierung von und zu *java.util.Date*

Trotz aller Nachteile ist das Java-Standard-Date unverzichtbar. Konvertierungen sind also unvermeidbar. Trotzdem sollte an erster Stelle immer der Versuch stehen, direkte Konvertierungen zu vermeiden:

- Wenn ein String bereits das Datum enthält, ist es meist günstiger, den Datumsteil nochmals zu parsen.
- Wenn das Date nur gebraucht wird, um es als String zu formatieren, kann man einen eigenen Formatter bauen, der z. B. ein Day-Objekt als Argument akzeptiert.

[3] Ein einfacher Test auf einem PC mit Intel 7-2600 CPU/3.40 GHz und 8 GB Speicher ergab im Interpretermodus (-Xint), dass die Konvertierung JD->Day->JD aller ca. 3 Mill. Werte des Gültigkeitsbereichs weniger als 25 Sekunden benötigt, die Konvertierung der Day-Werte in ein Date und zurück benötigt ca. 100 Sekunden (eine vorallokierte CalendarInstanz für alle) bis 130 Sekunden (ein Calendar pro Datum). Im Servermodus lauten die wegen der Ausführung in einer Schleife weniger aussagekräftigen Werte: 0,7 s, 3,0 s, 3,8 s.

- Manche Datenbanken bieten Funktionen zur Darstellung von Datumswerten als ganze Zahlen, die der hier vorgestellten Version weigehend entsprechen. Wenn das nicht der Fall ist, kann man Datumswerte als Zeichenketten transferieren.

Wenn die direkte Konvertierung unvermeidlich ist, erfolgt sie mit Hilfe eines GregorianCalendar-Objekts:

```
 Date date= …;
Calendar calendar= new GregorianCalendar(timezone);
 calendar.setTime(date);
 int jdToday= JulianDayConverter.calculateJD(
         calendar.get(Calendar.YEAR),
         calendar.get(Calendar.MONTH)+1,
         calendar.get(Calendar.DAY_OF_MONTH));
```

Nicht vergessen darf man dabei, die mit null beginnende Monatszählung in Java und selbstverständlich wird auch die Zeitzone explizit gesetzt.

Implizite Verwendung

Eine Variante der vorgestellten fortlaufenden Tagesnummer stellt die Verwendung einer unmittelbar lesbaren ganzen Zahl, z. B. 19700101 für 1.1.1970 dar. Sie ist leicht aus einem Day zu gewinnen:

```
int readableDay=
   day.getYear() * 10000 + day.getMonth() * 100 + day.getDay();
```

Umgekehrt erhält man eine Day-Instanz durch:

```
new Day(readableDay / 10000, (readableDay % 10000) / 100,
        readableDay % 100);
```

Der Nachteil der impliziten Verwendung ergibt sich daraus, dass viele Berechnungen eine Umwandlung in JD und anschließend zurück in das lesbare Format erfordern. In den meisten Fällen dürfte dieser zusätzlich Aufwand vernachlässigbar sein.

Alternativen

Wegen der relativ großen Werte der JD wurden zu Zeiten des 16-Bit-Integer-Formats oder vorausgehender ähnlich beschränkter BCD-Zahldarstellungen das Modifizierte Julianische Datum (MJD) eingeführt, dessen Startpunkt am 17.11.1858 liegt, wobei gilt: JD(x)=MJD(x)+2400000. Diese Einschränkung bringt auf heutigen Computern keinerlei Vorteile mehr. Das MJD wird hauptsächlich in den Geo-Wissenschaften und der Raumfahrt verwendet.

Grundsätzlich kann jeder beliebige Tag als Basis einer fortlaufenden Tageszählung genommen werden. Wie oben bereits erwähnt, verwendet COBOL den 1.1.1601 als Tag eins, einige Tabellenkalkulationsprogramme den 1.1.1900 bzw. 1.1.1904. Gerade das Beispiel Excel, das 1900 fälschlicherweise als Schaltjahr ansieht, zeigt aber, dass bei solchen willkürlich gewählten Stichtagen leicht Fehler unterlaufen können.

Der auf den ersten Blick naheliegende Gedanke, die Tageszahlen parallel zur UTC-Epoche vom 1.1.1970 aus zu berechnen, bietet ebenfalls keine wirklichen Vorteile. Stattdessen hat man es dann für Daten vor diesem Stichtag mit negativen Werten zu tun, was in der Handhabung unpraktischer ist. Eine Konvertierung von Zeitpunkten in Tageszahlen durch einfache Division ist zwar unter bestimmten Umständen möglich, birgt aber wegen der zu beachtenden Randbedingungen (Sommer-/Winterzeit, Schaltsekunden) erhebliche Risiken.

Die Julianische Tageszahl wird durch Erweiterung um einen gebrochenen Anteil, der die Tageszeit darstellt, zum Julianischen Datum, wie es vor allem in der Astronomie verwendet wird. Da es hier um reine Datumswerte geht, soll das aber nicht weiter betrachtet werden.

Erfahrungen

In zahlreichen Projekten hat sich gezeigt, dass Javas Date unangemessen ist, wenn wirklich nur ein Datum benötigt wird. Deshalb findet man immer wieder ad hoc-Lösungen, bei denen Tage von einem willkürlich gewählten Stichtag aus gezählt werden. Ohne klares Konzept sind solche Lösungen jedoch kritisch für die Stabilität des Codes. Insbesondere Konvertierungen in andere Typen (java.util.Date) oder Stringformate erweisen sich häufig als Schwachstellen.

Im Allgemeinen erlauben Tageszahlen effiziente und übersichtliche Programme. Vor allem der Wegfall von Berechnungen, die Kalenderklassen benutzen, kann eine erhebliche Leistungssteigerung bringen. Ein typisches Anwendungsbeispiel für das vorgeschlagene Konzept war die Optimierung von Geldautomatenbefüllungen. Auf der Basis der täglichen Entnahmen der letzten zwei Jahre war eine Prognose für die nächsten 60 Tage zu erstellen. Dabei mussten die Verschiebungen von Wochentagen, Ultimos und Feiertagen berücksichtigt werden. Allein durch Umstellung auf eine fortlaufende Tageszahl konnten die Berechnungen so beschleunigt werden, dass auf einen speziellen Batchprozess mit Ablage der Ergebnisse in der Datenbank verzichtet werden konnte und stattdessen die benötigte Prognose jeweils in Echtzeit ermittelt wurde. Neben diesen Leistungsvorteilen gewinnt der Code an Klarheit und wird dadurch leichter versteh- und wartbar.

Zusammenfassung

Im Gegensatz zu Software, die originär aus dem Geschäftsbereich stammt (COBOL, Excel usw.) bietet Java keine angemessenen Mittel zur Behandlung reiner Datumswerte. In diesem Beitrag wird ein Ansatz präsentiert, mit dessen Hilfe dieses Manko in Projekten überwunden werden kann. Als Grundlage dient die aus der Astronomie stammende Julianische Tageszahl. Für die Umrechnung stehen schnelle und stabile Algorithmen zur Verfügung. Eine Implementierung in Java wird vorgestellt, wobei es sich dabei naturgemäß nur um die elementaren Funktionen handelt, die in konkreten Projekten durch weitere (Convenience-) Methoden ergänzt werden können.

Neben der direkten Verwendung einer fortlaufenden Tageszahl wird die Variante einer nichtfortlaufenden, aber unmittelbar lesbaren Tageszahl diskutiert. Sie stellt in allen Fällen, in denen der resultierende Overhead tolerierbar ist, einen guten Kompromiss dar.
Abschließend soll noch einmal darauf hingewiesen werden, dass die Konzepte Datum, Zeit und Zeitpunkt trotz ihrer Alltäglichkeit nicht trivial sind und sorgfältig unterschieden werden müssen. Unter dieser Voraussetzung ist die beschriebene Behandlung von Werten des Typs Datum nicht nur sehr effizient implementierbar, sondern trägt zu klar strukturiertem Code bei.

Literaturverzeichnis

[1] Joshua J. Bloch: Java: The Good, the Bad, and the Ugly Parts, devoxx 2011, Antwerpen, http://www.devoxx.com/display/DV11/Java++The+Good%2C+the+Bad%2C+and+the+Ugly+Parts

[2] Julianisches Datum, http://de.wikipedia.org/wiki/Julianisches_Datum

[3] Kalender-Umrechner, http://www.heinrichbernd.de/calendar/index_html?mode=jd

[4] Oracle Call Interface Programmer's Guide, Part Number A96584-01, http://docs.oracle.com/cd/B10500_01/appdev.920/a96584/oci03typ.htm#421890

[5] Eli Billauer, Why MySQL's (SQL) DATETIME can and should be avoided, 2009, http://billauer.co.il/blog/2009/03/mysql-datetime-epoch-unix-time/

Anhang

Um die Anwendung der Julianischen Tageszahl zu illustrieren, werden hier einige Ausschnitte aus den Testfällen aufgeführt.

- Hin- und Rückrechnung, Vergleich mit Beispielwert aus [2]

```
Day d1= new Day(1990, 1, 1);
int jd= JulianDayConverter.calculateJD(d1);
assertEquals(2447893, jd);
Day day= JulianDayConverter.createDayFromJD(jd);
assertEquals(d1, day);
assertEquals(d1.toString(), day.toString());
assertEquals("1990-01-01", day.toString());
```

- Wochentagsermittlung

```
Day d1= new Day(2012, 8, 6);
int jd= JulianDayConverter.calculateJD(d1);
assertEquals(0, jd%7);  // ein Montag == 0
Day day= JulianDayConverter.createDayFromJD(jd);
assertEquals(d1, day);
assertEquals(d1.toString(), day.toString());
```

- Fristberechnung Tageswert

```
Day d1= new Day(2012, 7, -2); // == 28.6.2012
int jd= JulianDayConverter.calculateJD(d1);
assertEquals(2456107, jd);
Day day= JulianDayConverter.createDayFromJD(jd);
assertEquals(new Day(2012, 6, 28), day);
assertEquals("2012-06-28", day.toString());
```

- Fristberechnung Monatswert

```
Day d1= new Day(2011, 19, 8); // == 8.7.2012
int jd= JulianDayConverter.calculateJD(d1);
assertEquals(2456117, jd);
Day day= JulianDayConverter.createDayFromJD(jd);
assertEquals(new Day(2012, 7, 8), day);
assertEquals("2012-07-08", day.toString());
```

- Zahlbereichskontrolle

```
Day d1= new Day(9999, 99, 99);
int jd= JulianDayConverter.calculateJD(d1); // 5376199
assertTrue(jd < Integer.MAX_VALUE/4);
```

- Konvertierung bzgl. java.util.Date

```
Date date= new Date(1344925513600L);//Tue Aug 14 08:25:13 CEST 2012
Calendar calendar= new GregorianCalendar(
                                TimeZone.getTimeZone("Germany/Berlin"));
calendar.setTime(date);
int jdGermany= JulianDayConverter.calculateJD(
        calendar.get(Calendar.YEAR),
        calendar.get(Calendar.MONTH)+1,
        calendar.get(Calendar.DAY_OF_MONTH));
assertEquals("2012-08-14",
          JulianDayConverter.createDayFromJD(jdGermany).toString());
calendar.setTimeZone(TimeZone.getTimeZone("America/Los_Angeles"));
int jdPacific= JulianDayConverter.calculateJD(
                   calendar.get(Calendar.YEAR),
                   calendar.get(Calendar.MONTH)+1,
                   calendar.get(Calendar.DAY_OF_MONTH));
assertEquals("2012-08-13", JulianDayConverter.createDayFromJD(
                                              jdPacific).toString());
```

Der Autor

Jürgen Lampe ist IT-Berater bei der Agon Solutions in Frankfurt. Vor seiner Tätigkeit als Berater wirkte er als Hochschullehrer an der Technischen Universität Dresden. Seit mehr als 15 Jahren befasst er sich mit Design und Implementierung von Java-Anwendungen im Bankenumfeld. An Fachsprachen (DSL) und Werkzeugen für deren Implementierung ist er seit seiner Studienzeit interessiert.

Email: juergen.lampe@agon-solutions.de

Agon Solutions

Die Agon Solutions, 2004 gegründet, ist ein unabhängiges IT-Dienstleistungsunternehmen mit Firmensitz in Eschborn bei Frankfurt und weiteren Standorten in Hamburg und Berlin. Das branchenübergreifende Dienstleistungsportfolio von Agon umfasst das „Agon-proven IT-Consulting“, eine bewährte, herstellerneutrale IT-Beratung; sowie die „Agon-tailored IT-Solutions“, zu denen maßgeschneiderte, individuelle Softwareentwicklung, passgenaue Softwareintegration und effiziente Business Intelligence Lösungen gehören. Agon stellt sich mit professionellem Projektmanagement, proaktivem Anforderungsmanagement und änderbaren Softwarearchitekturen auf die individuellen Bedürfnisse seiner Kunden ein. In ausgewählten Branchen wie Banken, Versicherungen, Touristik/Aviation und Health Care bietet Agon gemeinsam mit seinen Partnern Lösungen, die auf fundiertem Geschäftsprozess-Know-how beruhen und speziell auf die jeweilige Branche zugeschnitten sind: die „Agon-tailored Business Solutions“. Die plattformübergreifende technologische Kompetenz bei Agon reicht von klassischen Mainframe-Architekturen bis hin zu modernen Java/JEE Web- und Portal-Architekturen. Zu den Referenzkunden von Agon Solutions gehören unter anderen die AOK Berlin-Brandenburg, die Commerzbank, die Deutsche Bank, die Deutsche Bank Bauspar, die Deutsche Börse, die Finanz Informatik und die Deutsche Lufthansa.

Frankfurter Strasse 71-75
D-65760 Eschborn
Telefon: +49 6196 80269 0
Telefax: +49 6196 80269 11
http://www.agon-solutions.de

Handelsregister Frankfurt HRB 58185
St.-Nr. 4022826171
Geschäftsführer: Udo Peters