1-2013 | Frühjahr | www.ijug.eu

Java aktuell

Praxis. Wissen. Networking. Das Magazin für Entwickler

Java aktuell

Java – Programmiersprache mit Tiefgang

Neu auf dem Markt
Eclipse 4, Seite 18
Geronimo 3.0, Seite 65

Mainframe
Modernisieren mit Java, Seite 27

Wissen für Entwickler
Garbage Collection, Seite 36
Scrum & Kanban, Seite 61

Java und Oracle
Kostenlose ADF-Version, Seite 45
PL/SQL-Logik in Java-Anwendungen, Seite 46
Debugging für Forms 11*g*, Seite 50

D: 4,90 EUR A: 5,60 EUR CH: 9,80 CHF Benelux: 5,80 EUR ISSN 2191-6977

iJUG Verbund

Sonderdruck

3 Editorial
Wolfgang Taschner

5 Das Java-Tagebuch
Andreas Badelt

8 Neues von der JavaOne 2012
Mylène Diacquenod

9 Java als Treiber für den Erfolg der Kunden
Wolfgang Taschner

11 Datum in Java
Jürgen Lampe

16 „Wir möchten nicht, dass Java eine Einzelunternehmens-Plattform wird …“ Interview mit Gil Tene, CTO von Azul Systems und Mitglied im Exekutiv-Komitee des Java Community Process (JCP)

18 Ein erster Blick auf Eclipse 4
Dr. Jonas Helming und Marc Teufel

22 Impressum

23 Immer hübsch der Reihe nach …
Uwe Sauerbrei

27 Modernisieren mit Java auf dem Mainframe
Marc Bauer und Tobias Leicher

30 Java und das Open Data Protocol
Klaus Rohe

34 Inserentenverzeichnis

35 „Ich bin sehr zufrieden mit der Sprache …“ Interview mit Dominik Dorn, Vorsitzender der Java Student User Group Wien (JSUG)

36 Garbage Collection im Java-Umfeld
Mathias Dolag, Prof. Dr. Peter Mandl und Christoph Pohl

45 Oracle bietet kostenlose ADF-Version
Detlef Müller

46 Der Tiger im Tank: PL/SQL-Logik in Java-Anwendungen optimal nutzen
Björn Christoph Fischer

50 Oracle Forms 11g – Debugging, Statusmeldungen und Maskensteuerung durch eine Erweiterung des Java-Timers
Frank Christian Hoffmann

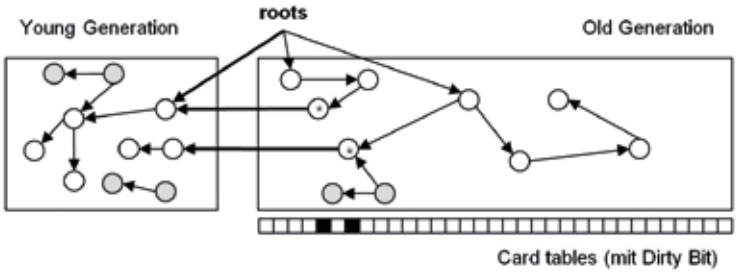
54 Linked Data – ein Web aus Daten
Angelo Veltens

58 Der Herbstcampus 2012 aus Sicht eines Besuchers
Steven Schwenke

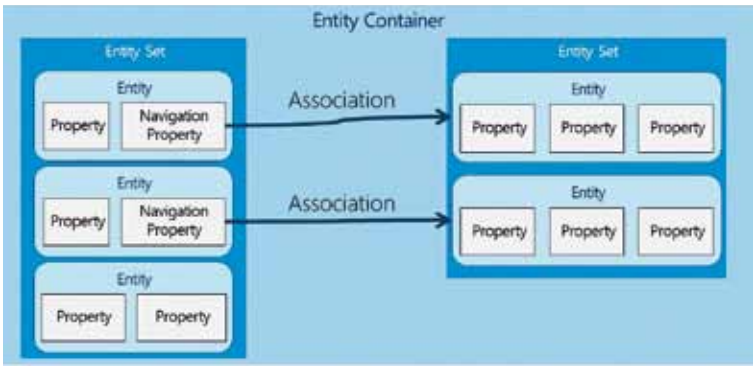
60 20 Jahre Java, 20 Folien, 20 Sekunden
Oliver Böhm

61 Scrum & Kanban in der Praxis
Martin Dilger

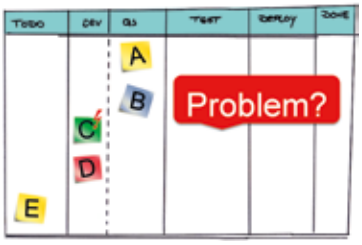
65 Geronimo 3.0 – modulare Hybride fahren gut
Frank Pientka



Intergenerational References bei der Garbage Collection, Seite 36



Das Entity Data Model im Open Data Protocol, Seite 30



Agiles Entwickeln, Seite 61

Das neue Framework, Seite 18

Datum in Java

Foto: Fotolia

Jürgen Lampe, A:gon Solutions GmbH

Java fehlt es bis heute an einem durchdachten Konzept für die Abbildung des Datums. Der ohne gründlichere 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 aufwändigen Lösungen.

Dass diese Klasse „java.util.Date" nicht besonders gut gelungen war, ist wohl auch den Entwicklern bei Sun recht schnell bewusst geworden. Inzwischen sind 22 der 33 öffentlichen („public") Methoden beziehungsweise Konstruktoren als überholt („deprecated") gekennzeichnet. Allerdings ist eine echte Aufarbeitung bisher leider unterblieben, was bedeutet, dass innerhalb der Java-Bibliotheken diese problematischen Funktionen weiterhin zum Einsatz kommen. 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 Zeitzonen-Einstellungen der Datenbank von denen der Java-Anwendung unterscheiden.

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 Kalender-Objekts ist aufwändig. Wenn Datums-Konvertierungen oder -Berechnungen sehr häufig nötig sind, braucht es spezielle Maßnahmen, um zu vermeiden, dass daraus ein Performance-Engpass 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 Standard-Einstellungen verlässt, weil SQL für „DATE" keine Zeitzone kennt (nur für „TIME" und „TIMESTAMP"). Die erforderliche explizite Behandlung etwa durch Aufruf der „getDate"-Methode mit „Calendar"-Parameter (seit JDBC 2.0) ist eine völlig unangemessene Verkomplizierung, die überdies Leistung kostet. Außerdem kann die im Allgemeinen unterschiedliche und nicht immer standardkonforme Implementierung des „DATE"-Typs von SQL in den verbreiteten Datenbanken zu weiteren Problemen führen.

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"), 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" [1].

Ein ganz spezielles Problem kann beim Remote Procedure Call (RPC) oder bei anderen Protokollen, die Objekt-Serialisierung verwenden, auftreten, wenn die beteiligten Seiten mit unterschiedlichen Zeitzonen-Einstellungen arbeiten. Weil Datumswerte in der Regel als „Zeitstempel" Mitternacht der entsprechenden Zeitzone realisieren, 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" [2]. Das ist zwar keine prinzipielle Frage, aber die Veränderbarkeit hat an vielen Stellen für Verwirrung gesorgt.

Ein Versuch, Unzulänglichkeiten bei der Zeitbehandlung in Java zu beheben, ist das Joda-Time-Projekt [3]. Ebenso wie beim daraus hervorgegangenen JSR 310 besteht das Ziel darin, eine funktionale Alternative für die Standard-Funktionen rund um „java.util.Date" zu bieten. Schwerpunkte sind eine verständlichere API und

bessere Performance. Der Grundgedanke, Datum und Uhrzeit gemeinsam zu verwalten, also das Datum letztlich aus einem Zeitpunkt abzuleiten, bleibt aber unverändert.

Eine echte Datumsklasse in Java

Wichtig ist es, bereits beim 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 unterscheiden, ob etwas täglich zur gleichen Uhrzeit oder alle 24 Stunden ausgeführt werden soll. „java.util.Timer. schedule(TimerTask task, Date firstTime, long period)" erledigt das Letztere. Was im allgemeinen Verständnis zunächst nach einer 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 unnötig 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, ist unbedingt zu prüfen, 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

```
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;
   }
}
```

Listing 1

```
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);
  }
}
```

Listing 2

```
  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));
```

Listing 3

schreiben. Diese Form hat Vorteile, wenn auch auf der Datenbank Berechnungen auf Tages-Basis ausgeführt werden sollen. Als Alternative bietet sich ein Abspeichern als „String“, etwa in der Form „YYYY-MM-DD“ an. Das erfordert zwar vor Berechnungen jeweils eine Konvertierung, ist aber für den Menschen leichter lesbar.

Eine weitere Frage, die möglichst früh geklärt werden muss, ist die nach dem benötigten Bereich gültiger Datumswerte. Ein erheblicher Teil des Aufwands bei den Kalender-Berechnungen in Java wird durch die Berücksichtigung von Datums-Umstellungen (Julianisch nach Gregorianisch) und Zeitzonen-Veränderungen 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. Januar 1583 bis zum 31. Dezember 9999.

Die Julianische Tageszahl

Prinzipiell kann man natürlich jedes beliebige Datum auswählen, um davon beginnend die Tage zu zählen. Es erleichtert aber die Kommunikation und das Verständnis, wenn man auf eine Konvention setzt. In diesem Artikel wird das chronologische Julianische Datum als Berechnungsbasis verwendet. Der Name ist etwas irreführend, da es keinen unmittelbaren Bezug zum Julianischen Kalender gibt [4]. Vorteil dieser Wahl ist, dass die benötigten Algorithmen erprobt und aufbereitet verfügbar sind. Überdies gibt es Online-Tools zur Berechnung [wie 5], die den Test erleichtern.

Die in [4] angegebenen Umrechnungs-Algorithmen sind stabil gegenüber Wertüberschreitungen, sodass 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- oder Tagesfrist kann damit sehr einfach berechnet werden, indem der entsprechende Ausgangswert einfach um die Frist vergrößert und anschließend eine Hin- und Rückkonvertierung ausgeführt wird.

Implementierung

Es folgt nun die exemplarische Implementierung der ausgewählten Strategie. Wir beginnen mit einer Containerklasse für das Datum (siehe 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, das heißt „new Day(2012, 8, 0).equals(new Day(2012, 7, 31))“ ist immer falsch.

Die Umrechnung zwischen „Day“ beziehungsweise den Integer-Werten für „Jahr“, „Monat“ und „Tag“ und der Julianischen Tageszahl (JD) erfolgt durch statische Methoden der Klasse „JulianDayConverter“ (siehe Listing 2). Die Algorithmen wurden aus [4] übernommen und lediglich so umgeformt, dass sie mit Integer-Arithmetik berechnet werden können. Für ihre Herleitung wird auf die dort angegebene Literatur verwiesen. Der Code enthält noch keine Überprüfungen darauf, ob die Beschränkung auf das Zeitintervall „1.1.1583“ bis „31.12.9999“ eingehalten ist. Es hängt vom konkreten Projekt ab, ob ein solcher Test wirklich erforderlich ist.

Bei Bedarf kann die Umrechnung auf Julianische Daten (also 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 (siehe Listing 1 und 2).

Vorteile

Im Kasten auf Seite 15 finden sich einige Beispiele für die Verwendung der Methoden. Die Vorteile der Julianischen Tageszahl sind klar ersichtlich:

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

Listing 4

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

Listing 5

- 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)
- Einfache String-Konvertierung

Die Umrechnung erfolgt durch statische Methoden, die auch intern keinerlei Objekte anlegen. Das ist ein wichtiger Unterschied zur Verwendung der „Calendar"-Klasse in Java, die aufwändig 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.

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 circa drei Millionen Werte des Gültigkeitsbereichs weniger als 25 Sekunden benötigt, die Konvertierung der „Day"-Werte in ein „Date" und zurück hingegen etwa 100 Sekunden (eine vorallokierte „Calendar"-Instanz für alle) bis 130 Sekunden (ein „Calendar" pro Datum). Im Server-Modus lauten die bei der Ausführung in einer Schleife weniger aussagekräftigen Werte 0,7 s, 3,0 s und 3,8 s.

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 nochmal zu parsen.
- Wenn das Date nur gebraucht wird, um es als String zu formatieren, kann man einen eigenen Formatter bauen, der beispielsweise ein „Day"-Objekt als Argument akzeptiert.
- Manche Datenbanken bieten Funktionen zur Darstellung von Datumswerten als ganze Zahlen, die der hier vorgestellten Version weitgehend entsprechen. Wenn das nicht der Fall ist, kann man Datumswerte als Zeichenketten transferieren.

Wenn die direkte Konvertierung unvermeidlich ist, erfolgt sie mithilfe eines „GregorianCalendar"-Objekts (siehe Listing 3). 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, etwa „19700101" für „1.1.1970", dar. Sie ist leicht aus einem „Day" zu gewinnen (siehe Listing 4). Listing 5 zeigt, wie man umgekehrt eine „Day"-Instanz erhält. 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 zusätzliche Aufwand vernachlässigbar sein.

Alternativen

Wegen der relativ großen Werte der „JD" wurde zu Zeiten des 16-Bit-Integer-Formats oder vorausgehender ähnlich beschränkter „BCD"-Zahldarstellungen das Modifizierte Julianische Datum (MJD) eingeführt, dessen Startpunkt am 17. November 1858 liegt, wobei gilt: JD(x) = MJD(x) + 2400000. Diese Einschränkung bringt auf heutigen Computern keinerlei Vorteile mehr. Das „MJD" kommt hauptsächlich in den Geo-Wissenschaften und der Raumfahrt zum Einsatz.

Grundsätzlich kann jeder beliebige Tag als Basis einer fortlaufenden Tageszählung genommen werden. Wie bereits erwähnt, verwendet Cobol den 1. Januar 1601 als Tag „eins", einige Tabellenkalkulationsprogramme den 1. Januar 1900 beziehungsweise den 1. Januar 1904. Gerade das Beispiel „Excel", das 1900 fälschlicherweise als Schaltjahr ansieht, zeigt, 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. Januar 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, das dann wieder ein Zeitpunkt ist. Es wird vor allem in der Astronomie verwendet.

Praktische Erfahrungen

In zahlreichen Projekten hat sich gezeigt, dass „Date" in Java 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 String-Formate erweisen sich häufig als Schwachstellen.

Im Allgemeinen erlauben Tageszahlen effiziente und übersichtliche Programme. Vor allem der Wegfall von Berechnungen, die Kalender-Klassen benutzen, kann eine erhebliche Leistungssteigerung bringen.

Testfälle

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

```
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());
```

Hin- und Rückrechnung, Vergleich mit Beispielwert aus [4]

```
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 Tageswert

```
Date date= new Date(13449255136000L);
      //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());
```

Konvertierung bezüglich „java.util.Date"

```
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());
```

Wochentagsermittlung

```
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());
```

Fristberechnung Monatswert

Ein typisches Anwendungsbeispiel für das vorgeschlagene Konzept war die Optimierung von Geldautomaten-Befüllungen. Auf der Basis der täglichen Entnahmen der letzten zwei Jahre war eine Prognose für die nächsten sechzig Tage zu erstellen. Dabei waren die Verschiebungen von Wochentagen, Ultimos und Feiertagen zu berücksichtigen. Allein durch Umstellung auf eine fortlaufende Tageszahl konnten die Berechnungen so beschleunigt werden, dass auf einen speziellen Batch-Prozess mit Ablage der Ergebnisse in der Datenbank verzichtet werden konnte und stattdessen die benötigte Prognose jeweils in Echtzeit zur Verfügung stand. Neben diesen Leistungsvorteilen gewinnt der Code an Klarheit und wird dadurch leichter versteh- und wartbar.

Fazit

Im Gegensatz zu Software, die originär aus dem Geschäftsbereich stammt (COBOL, Excel etc.), bietet Java keine angemessenen Mittel zur Behandlung reiner Datumswerte. Dieser Artikel liefert einen Ansatz, mit dessen Hilfe sich dieses Manko in Projekten überwinden lässt. 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 nicht fortlaufenden, aber unmittelbar lesbaren Tageszahl diskutiert. Sie stellt in allen Fällen, in denen der resultierende Overhead tolerierbar ist, einen guten Kompromiss dar.

Abschließend sei nochmal darauf hingewiesen, 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.

Quellen

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

[2] 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

[3] Joda Time – Java date and time API: http://joda-time.sourceforge.net

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

[5] Kalender-Umrechner: http://www.heinrich-bernd.de/calendar/index_html?mode=jd

Jürgen Lampe
juergen.lampe@agons-solutions.de

Dr. Jürgen Lampe ist IT-Berater bei der A:gon Solutions GmbH in Frankfurt. Vor seiner Tätigkeit als Berater wirkte er als Hochschullehrer an einer Technischen Universität. 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.

www.ijug.eu

JETZT ABO BESTELLEN

Sichern Sie sich 4 Ausgaben für 18 EUR

Für Oracle-Anwender und Interessierte gibt es das Java aktuell Abonnement auch mit zusätzlich sechs Ausgaben im Jahr der Fachzeitschrift DOAG *News* und vier Ausgaben im Jahr Business News zusammen für 70 EUR. Weitere Informationen unter www.doag.org/shop/

FAXEN SIE DAS AUSGEFÜLLTE FORMULAR AN

0700 11 36 24 39

ODER BESTELLEN SIE ONLINE

go.ijug.eu/go/abo

Interessenverbund der Java User Groups e.V.
Tempelhofer Weg 64
12347 Berlin

Java aktuell

+++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN +++ AUSSCHNEIDEN +++ ABSCHICKEN +++ AUSFÜLLEN

Ja, ich bestelle das Abo Java aktuell – das IJUG-Magazin: 4 Ausgaben zu 18 EUR/Jahr

Ja, ich bestelle den kostenfreien Newsletter: Java aktuell – der iJUG-Newsletter

ANSCHRIFT

Name, Vorname

Firma

Abteilung

Straße, Hausnummer

PLZ, Ort

GGF. RECHNUNGSANSCHRIFT

Straße, Hausnummer

PLZ, Ort

E-Mail

Telefonnummer

X Die allgemeinen Geschäftsbedingungen* erkenne ich an, Datum, Unterschrift

*Allgemeine Geschäftsbedingungen:

Zum Preis von 18 Euro (inkl. MwSt.) pro Kalenderjahr erhalten Sie vier Ausgaben der Zeitschrift "Java aktuell - das iJUG-Magazin" direkt nach Erscheinen per Post zugeschickt. Die Abonnementgebühr wird jeweils im Januar für ein Jahr fällig. Sie erhalten eine entsprechende Rechung. Abonnementverträge, die während eines Jahres beginnen, werden mit 4,90 Euro (inkl. MwSt.) je volles Quartal berechnet. Das Abonnement verlängert sich automatisch um ein weiteres Jahr, wenn es nicht bis zum 31. Oktober eines Jahres schriftlich gekündigt wird. Die Wiederrufsfrist beträgt 14 Tage ab Vertragserklärung in Textform ohne Angabe von Gründen.