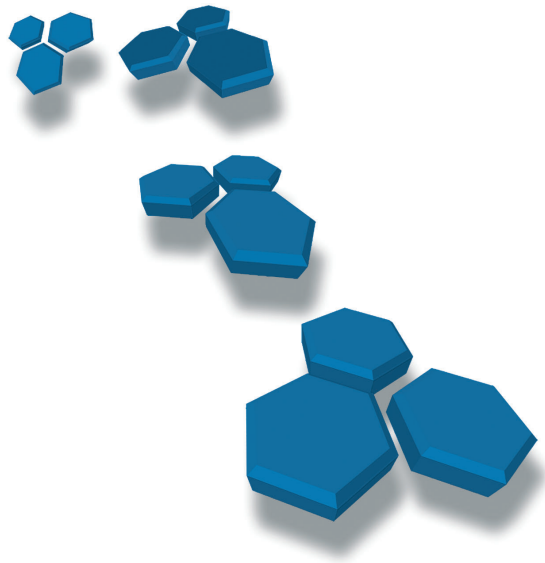
PEARSON Education

ADDISON-WESLEY

Dirk Louis, Peter Müller

Das Java Codebook



ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Datum und Uhrzeit

39 Aktuelles Datum abfragen

Der einfachste und schnellste Weg, das aktuelle Datum abzufragen, besteht darin, ein Objekt der Klasse `Date` zu erzeugen:

```
import java.util.Date;

Date today = new Date();
System.out.println(today);
```

Ausgabe:

```
Thu Mar 31 10:54:31 CEST 2005
```

Wenn Sie dem Konstruktor keine Argumente übergeben, ermittelt er die aktuelle Systemzeit als Anzahl Millisekunden, die seit dem 01.01.1970 00:00:00 Uhr, GMT, vergangen sind, und speichert diese in dem `Date`-Objekt. Wenn Sie das `Date`-Objekt mit `println()` ausgeben (oder in einen String einbauen), wird seine `toString()`-Methode aufgerufen, die aus der Anzahl Millisekunden das Datum berechnet. Und genau hier liegt das Problem der `Date`-Klasse.

Die `Date`-Klasse arbeitet nämlich intern mit dem Gregorianischen Kalender. Dieser ist zwar weit verbreitet und astronomisch korrekt, jedoch bei weitem nicht der einzige Kalender. Bereits im JDK 1.1 wurden der Klasse `Date` daher die abstrakte Klasse `Calendar` und die von `Calendar` abgeleitete Klasse `GregorianCalendar` an die Seite gestellt. Die Idee dahinter:

- Neben `GregorianCalendar` können weitere Klasse für andere Kalender implementiert werden.
- Von der statischen Methode `Calendar.getInstance()` kann sich der Programmierer automatisch den passenden Kalender zur Lokale des aktuellen Systems zurückliefern lassen.

Leider gibt es derzeit nur zwei vordefinierte Kalenderklassen, die von `getInstance()` zurückgeliefert werden: `sun.util.BuddhistCalendar` für die Thai-Lokale und `GregorianCalendar` für alle anderen Lokalen.

Trotzdem sollten Sie den Empfehlungen von Sun folgen und die Klasse `Date` nur dann verwenden, wenn Sie an der reinen Systemzeit interessiert sind oder die chronologische Reihenfolge verschiedener Zeiten prüfen wollen. Wenn Sie explizit mit Datumswerten programmieren müssen, verwenden Sie `Calendar` oder `GregorianCalendar`.

```
// Aktuelles Datum mit Calendar abfragen
import java.util.Calendar;

Calendar calendar = Calendar.getInstance();
System.out.println(
      java.text.DateFormat.getDateTimeInstance().format(calendar.getTime()) );
```

Der Aufruf `Calendar.getInstance()` liefert ein Objekt einer `Calendar`-Klasse zurück (derzeit für nahezu alle Lokalen eine `GregorianCalendar`-Instanz, siehe oben). Das `Calendar`-Objekt repräsentiert die aktuelle Zeit (Datum und Uhrzeit) gemäß der auf dem System eingestellten Lokale und Zeitzone.

Die Felder (Jahr, Monat, Stunde ...) eines `Calendar`-Objekts können mit Hilfe der get-/set-Methoden der Klasse abgefragt bzw. gesetzt werden.

Methode	Beschreibung

`int get(int field)`	Zum Abfragen der verschiedenen Feldwerte. Die Felder werden durch folgende Konstanten ausgewählt:
	`AM_PM // AM (Vormittag) oder PM (Nachmittag)`
	`DATE // entspricht DAY_OF_MONTH`
	`DAY_OF_MONTH // Tag im Monat, beginnend mit 1`
	`DAY_OF_WEEK // Tag in Woche (1 (SUNDAY) - 7 (SATURDAY))`
	`DAY_OF_WEEK_IN_MONTH // 7-Tage-Abschnitt in Monat, beginnend mit 1`
	`DAY_OF_YEAR // Tag im Jahr, beginnend mit 1`
	`DST_OFFSET // Sommerzeitverschiebung in Millisekunden`
	`ERA // vor oder nach Christus`
	`HOUR // Stunde vor oder nach Mittag (0 - 11)`
	`HOUR_OF_DAY // Stunde (0 - 23)`
	`MILLISECOND // Millisekunden (0-999)`
	`MINUTE // Minuten (0-59)`
	`MONTH // Monat, beginnend mit JANUARY`
	`SECOND // Sekunde (0-59)`
	`WEEK_OF_MONTH // Woche in Monat, beginnend mit 0`
	`WEEK_OF_YEAR // Woche in Jahr, beginnend mit 1`
	`YEAR // Jahr`
	`ZONE_OFFSET // Verschiebung für Zeitzone in Millisekunden`

`Date getTime()`	Liefert das Datum als `Date`-Objekt zurück.
`long getTimeInMillis()`	Liefert das Datum als Millisekunden seit/bis zum 01.01.1970 00:00:00 Uhr, GMT zurück.
`void set(int field, int value)`	Setzt den angegebenen Feldwert. Zur Bezeichnung der Felder siehe `get()`.

`void set(int year, int month, int date)`
`void set(int year, int month, int date, int hourOfDay, int minute)`
`void set(int year, int month, int date, int hourOfDay, int minute, int second)`
	Setzt Jahr, Monat (0-11) und Tag (1-31). Optional können auch noch Stunde (0-23), Minute und Sekunde angegeben werden.

`void setTime(Date d)`	Setzt das Datum gemäß dem übergebenen `Date`-Objekt.
`void setTimeInMillis(long millis)`	Setzt das Datum gemäß der übergebenen Anzahl Millisekunden seit/bis zum 01.01.1970 00:00:00 Uhr, GMT.

Tabelle 15: Get-/Set-Methoden zum Abfragen und Setzen der Datumsfelder der Calendar-Klasse

Achtung

Die Klasse `Date` enthält ebenfalls Methoden zum Abfragen und Setzen der einzelnen Datums- und Zeitfelder. Diese sind jedoch als »deprecated« eingestuft, von ihrem Gebrauch wird abgeraten.

40 Bestimmtes Datum erzeugen

Es gibt verschiedene Wege, ein Objekt für ein bestimmtes Datum zu erzeugen.

Handelt es sich um ein Datum im Gregorianischen Kalender, können Sie direkt ein Objekt der Klasse `GregorianCalendar` erzeugen und dem Konstruktor Jahr, Monat (0–11) und Tag (1–31) übergeben:

```
import java.util.GregorianCalendar;
...
Calendar birthday = new GregorianCalendar(1964, 4, 20);
```

Andere Kalender werden (mit Ausnahme des Buddhistischen und des Julianischen Kalenders, siehe unten) derzeit nicht unterstützt. Sie können Ihren Code aber bereits so aufsetzen, dass in Zukunft, wenn lokale `Calendar`-Implementierungen verfügbar sind und von `Calendar.getInstance()` unterstützt werden, das Datum nach dem lokalen Kalender verarbeitet wird. In diesem Fall lassen Sie sich von `Calendar.getInstance()` ein Objekt des lokalen Kalenders zurückliefern und ändern das von diesem Objekt repräsentierte Datum durch Setzen der Felder für Jahr, Monat und Tag:

```
import java.util.Calendar;
...
Calendar birthday = Calendar.getInstance();
birthday.set(1964, 4, 20);
```

Achtung

Für Daten vor dem 15. Oktober 1582 berechnet die Klasse `GregorianCalendar` das Datum nach dem Julianischen Kalender. Dies ist sinnvoll, da an diesem Tag – der dem 5. Oktober 1582 im Julianischen Kalender entspricht – der Gregorianische Kalender erstmals eingeführt wurde (in Spanien und Portugal). Andere Länder folgten nach und nach. In England und Amerika begann der Gregorianische Kalender beispielsweise mit dem 14. September 1752. Wenn Sie die Lebensdaten englischer bzw. amerikanischer Persönlichkeiten oder Daten aus der englischen bzw. amerikanischen Geschichte, die vor der Einführung des Gregorianischen Kalender liegen, historisch korrekt darstellen möchten, müssen Sie das Datum der Einführung mit Hilfe der Methode `setGregorianChange(Date)` umstellen.

```
GregorianCalendar change = new GregorianCalendar(1752, 8, 14, 1, 0, 0);
((GregorianCalendar) birthday).setGregorianChange(change.getTime());
```

Wenn Sie Interesse halber beliebige Daten nach dem Gregorianischen Kalender berechnen möchten, rufen Sie `setGregorianChange(Date(Long.MIN_VALUE))` auf. Wenn Sie beliebige Daten nach dem Julianischen Kalender berechnen wollen, rufen Sie `setGregorianChange(Date(Long.MAX_VALUE))` auf.

Der Vollständigkeit halber sei erwähnt, dass es auch die Möglichkeit gibt, ein `Date`-Objekt durch Angabe von Jahr (abzgl. 1900), Monat (0-11) und Tag (1-31) zu erzeugen:

```
Date birthday1 = new Date(64, 4, 20);
```

Vom Gebrauch dieses Konstruktors wird allerdings abgeraten, er ist als deprecated markiert.

Exkurs

Der Gregorianische Kalender und die Klasse GregorianCalendar

Vor der Einführung des Gregorianischen Kalenders im Jahre 1582 durch Papst Gregor XIII. galt in Europa der Julianische Kalender. Der Julianische Kalender, von dem ägyptischen Astronomen Sosigenes ausgearbeitet und von Julius Cäsar im Jahre 46 v. Chr. in Kraft gesetzt, war ein reiner Sonnenkalender, d.h., er richtete sich nicht nach den Mondphasen, sondern nach der Länge des mittleren Sonnenjahres, die Sosigenes zu 365,25 Tagen berechnete. Der Julianische Kalender übernahm die zwölf römischen Monate, korrigierte aber deren Längen auf die noch heute gültige Anzahl Tage, so dass das Jahr fortan 365 Tage enthielt. Um die Differenz zum »angenommenen« Sonnenjahr auszugleichen, wurde *alle* vier Jahre ein Schaltjahr eingelegt.

Tatsächlich ist das mittlere Sonnenjahr aber nur 365,2422 Tage lang (tropisches Jahr). Der Julianische Kalender hinkte seiner Zeit also immer weiter hinterher, bis im Jahre 1582 das Primar-Äquinoktium auf den 11. statt den 21. März fiel.

Um die Differenz auszugleichen, verfügte Papst Gregor XIII. im Jahr 1582, dass in diesem Jahr auf den 4. Oktober der 15. Oktober folgen sollte. Gleichzeitig wurde der Gregorianische Kalender eingeführt, der sich vom Julianischen Kalender in der Berechnung der Schaltjahre unterscheidet. Während der Julianische Kalender *alle* vier Jahre ein Schaltjahr einlegte, sind im Gregorianischen Kalender alle Jahrhundertjahre, die nicht durch 400 teilbar sind, keine Schaltjahre. Durch diese verbesserte Schaltregel ist ein Jahr im Gregorianischen Kalender durchschnittlich 365,2425 Tage lang, was dem tatsächlichen mittleren Wert von 365,2422 Tagen (tropisches Jahr) sehr nahe kommt. (Erst nach 3000 Jahren wird sich die Abweichung zu einem Tag addieren.)

Spanien, Portugal und Teile Italiens führten den Gregorianischen Kalender wie vom Papst vorgesehen in der Nacht vom 4. auf den 5./15. Oktober ein. Die meisten katholischen Länder folgten in den nächsten Jahren, während die protestantischen Länder den Kalender aus Opposition zum Papst zunächst ablehnten. Die orthodoxen Länder Osteuropas führten den Gregorianischen Kalender gar erst im 20. Jahrhundert ein.

Land	Einführung
Spanien, Portugal, Teile Italiens	04./15. Oktober 1582
Frankreich	09./20. Dezember 1582
Bayern	05./16. Oktober 1583
Hzm. Preußen	22. August/02. September 1612
England, Amerika	02./14. 1752
Schweden	17. Februar/1. März 1753
Russland	31. Januar/14. Februar 1918
Griech.-Orthodoxe Kirche	10./24. März 1924
Türkei	1927

Tabelle 16: Einführung des Gregorianischen Kalenders

Die Klasse `GregorianCalendar` implementiert eine Hybridform aus Gregorianischem und Julianischem Kalender. Anhand des Datums der Einführung des Gregorianischen Kalenders interpretiert sie Datumswerte entweder als Daten im Gregorianischen oder Julianischen Kalender (siehe Hinweis weiter oben). Das Datum der Einführung kann mit Hilfe der Methode `setGregorianChange(Date d)` angepasst werden.

Das Datum, das eine `GregorianCalendar`-Instanz repräsentiert, kann durch Angabe der Datumsfelder (Jahr, Monat, Tag ...), als `Date`-Objekt oder als Anzahl Millisekunden seit/bis zum 01.01.1970 00:00:00 Uhr, GMT, festgelegt und umgekehrt auch als Werte der Datumsfelder, `Date`-Objekt oder Anzahl Millisekunden abgefragt werden. Für die korrekte Umrechnung zwischen Datumsfeldern und Anzahl Millisekunden sorgen dabei die von `Calendar` geerbten und in `GregorianCalendar` überschriebenen `protected`-Methoden `computeFields()` und `computeTime()`.

41 Datums-/Zeitangaben formatieren

Zur Formatierung von Datums- und Zeitangaben gibt es drei Wege zunehmender Komplexität, aber auch wachsender Gestaltungsfreiheit:

- `toString()`
- `DateFormat`-Stilen
- `SimpleDateFormat`-Muster

Formatierung mit toString()

Die einfachste Form der Umwandlung einer Datums-/Zeitangabe in einen String bietet die `toString()`-Methode. Ist die Datums-/Zeitangabe in ein `Date`-Objekt verpackt, erhält man auf diese Weise einen String aus (engl.) Wochentagskürzel, (engl.) Monatskürzel, Tag im Monat, Uhrzeit, Zeitzone und Jahr:

```
Thu Mar 31 10:54:31 CEST 2005
```

Wer Gleiches von der `toString()`-Methode der Klasse `Calendar` erwartet, sieht sich allerdings getäuscht. Die Methode ist rein zum Debuggen gedacht und packt in den zurückgelieferten String alle verfügbaren Informationen über den aktuellen Zustand des Objekts. Um dennoch einen vernünftigen Datums-/Zeit-String zu erhalten, müssen Sie sich die im `Calendar`-Objekt gespeicherte Zeit als `Date`-Objekt zurückliefern lassen und dessen `toString()`-Methode aufrufen:

```
Calendar calendar = Calendar.getInstance();
Date today = calendar.getTime();
System.out.println(date);
```

Ausgabe:

```
Thu Mar 31 10:54:31 CEST 2005
```

Formatierung mit DateFormat-Stilen

Die Klasse `DateFormat` definiert vier vordefinierte Stile zur Formatierung von Datum und Uhrzeit: `SHORT`, `MEDIUM` (= `DEFAULT`), `LONG` und `FULL`.

Die Klasse `DateFormat` selbst ist abstrakt, definiert aber verschiedene statische Factory-Methoden, die passende Objekte abgeleiteter Klassen (derzeit nur `SimpleDateFormat`) zur Formatierung von Datum, Uhrzeit oder der Kombination aus Datum und Uhrzeit zurückliefern.

Die Formatierung mit `DateFormat` besteht daher aus zwei Schritten:

1. Sie rufen die gewünschte Factory-Methode auf und lassen sich ein Formatierer-Objekt zurückliefern.

```
import java.text.DateFormat;

// Formatierer für reine Datumsangaben im SHORT-Stil
DateFormat df = DateFormat.getDateInstance(DateFormat.SHORT);
```

2. Sie übergeben die Datums-/Zeitangabe als `Date`-Objekt an die `format()`-Methode des Formatierers und erhalten den formatierten String zurück.

```
Calendar calendar = Calendar.getInstance();
String str = df.format(calendar.getTime());
```

Die Klasse `DateFormat` definiert vier Factory-Methoden, die gemäß der auf dem System eingestellten Lokale formatieren:

```
getInstance()           // Formatierer für Datum und Uhrzeit im SHORT-Stil
getDateInstance()       // Formatierer für Datum im DEFAULT-Stil (= MEDIUM)
getTimeInstance()       // Formatierer für Uhrzeit im DEFAULT-Stil (= MEDIUM)
getDateTimeInstance()   // Formatierer für Datum und Uhrzeit
                        // im DEFAULT-Stil (= MEDIUM)
```

Die letzten drei Methoden sind zweifach überladen, so dass Sie einen anderen Stil bzw. Stil und Lokale vorgeben können, beispielsweise:

```
getDateInstance(int stil)
getDateInstance(int stil, Locale loc)
```

gibt eine Übersicht über die Formatierung durch die verschiedenen Stile.

Stil	Formatierung (Lokale de_DE)
Datum	
`SHORT`	31.03.05
`MEDIUM` (= `DEFAULT`)	31.03.2005
`LONG`	31. März 2005
`FULL`	Donnerstag, 31. März 2005
Uhrzeit	
`SHORT`	19:51
`MEDIUM` (= `DEFAULT`)	19:51:14
`LONG`	19:51:14 CEST
`FULL`	19.51 Uhr CEST

Tabelle 17: DateFormat-Stile

Hinweis

Zur landesspezifischen Formatierung mit Lokalen siehe auch Rezepte in Kategorie »Internationalisierung«.

Formatierung mit SimpleDateFormat-Mustern

Wer mit den vordefinierten `DateFormat`-Formatstilen nicht zufrieden ist, kann sich mit Hilfe der abgeleiteten Klasse `SimpleDateFormat` einen individuellen Stil definieren. `SimpleDateFormat` besitzt einen Konstruktor

```
SimpleDateFormat(String format, Locale loc)
```

der neben der Angabe der Lokale auch einen Formatstring erwartet. Dieser String enthält feste datums- und zeitrelevante Formatanweisungen und darf beliebig durch weitere Zeichenfolgen, die in `' '` eingeschlossen sind, unterbrochen sein.

```
SimpleDateFormat df = new SimpleDateFormat("'Heute ist der 'dd'. 'MMMM");
```

Dieser Aufruf erzeugt eine Ausgabe der Art:

```
"Heute ist der 12. Juni".
```

Die wichtigsten Formatanweisungen für Datum und Zeit lauten:

Format	Beschreibung
`G`	»v. Chr.« oder »n. Chr« (in englischsprachigen Lokalen »BC« oder »AD«)
`yy`	Jahr, zweistellig
`yyyy`	Jahr, vierstellig
`M, MM`	Monat, einstellig (soweit möglich) bzw. immer zweistellig (01, 02 ...)
`MMM`	Monat als 3-Buchstaben-Kurzform
`MMMM`	voller Monatsname
`w, ww`	Woche im Jahr, einstellig (soweit möglich) bzw. immer zweistellig (01, 02 ...)
`W`	Woche im Monat
`D, DD, DDD`	Tag im Jahr, einstellig bzw. zweistellig (soweit möglich) oder immer dreistellig (001, 002 ...)
`d, dd`	Tag im Monat, einstellig (soweit möglich) bzw. immer zweistellig (01, 02 ...)
`E`	Wochentag-Kürzel: »Mo«, »Di«, »Mi«, »Do«, »Fr«, »Sa«, »So« (für englischsprachige Lokale werden dreibuchstabige Kürzel verwendet)
`EEEE`	Wochentag (ausgeschrieben)
`a`	»AM« oder »PM«
`H, HH`	Stunde (0-23), einstellig (soweit möglich) bzw. immer zweistellig (00, 01 ...)
`h, hh`	Stunde (1-12), einstellig (soweit möglich) bzw. immer zweistellig (00, 01 ...)
`K, KK`	Stunde (1-24), einstellig (soweit möglich) bzw. immer zweistellig (00, 01 ...)
`k, kk`	Stunde (1-12), einstellig (soweit möglich) bzw. immer zweistellig (00, 01 ...)

Tabelle 18: SimpleDateFormat-Formatanweisungen

Format	Beschreibung
m, mm	Minuten, einstellig (soweit möglich) bzw. immer zweistellig (00, 01 ...)
s, ss	Sekunden, einstellig (soweit möglich) bzw. immer zweistellig (00, 01 ...)
S, SS, SSS	Millisekunden, einstellig bzw. zweistellig (soweit möglich) oder immer dreistellig (001, 002 ...)
z	Zeitzone
Z	Zeitzone (gemäß RFC 822)

Tabelle 18: SimpleDateFormat-Formatanweisungen (Forts.)

42 Datumseingaben einlesen und auf Gültigkeit prüfen

Zum Einlesen von Datumseingaben benutzt man am besten eine der `parse()`-Methoden von `DateFormat`:

```
Date parse(String source)
Date parse(String source, ParsePosition pos)
```

So wie die `format()`-Methode von `DateFormat` ein `Date`-Objekt anhand der eingestellten Lokale und dem ausgewählten Pattern in einen String formatiert, analysieren die `parse()`-Methoden einen gegebenen String, ob er ein Datum enthält, das Lokale und Muster entspricht. Wenn ja, liefern sie das Datum als `Date`-Objekt zurück. Enthält der übergebene String keine passende Datumsangabe, löst die erste Version eine `ParseException` aus. Die zweite Version, welche ab der Position `pos` sucht, liefert `null` zurück.

Der folgende Code liest deutsche Datumseingaben im MEDIUM-Format (TT.MM.JJJJ) ein und gibt sie zur Kontrolle im FULL-Format aus. Für Ein- und Ausgabe werden daher unterschiedliche `DateFormat`-Instanzen (`parser` und `formatter`) erzeugt:

```
Date date = null;
DateFormat parser =
               DateFormat.getDateInstance(DateFormat.MEDIUM, Locale.GERMANY);
DateFormat formatter =
               DateFormat.getDateInstance(DateFormat.FULL, Locale.GERMANY);

try {
    // Datum aus Kommandozeile einlesen
    date = parser.parse(args[0]);

    // Datum auf Konsole ausgeben
    System.out.println(" " + formatter.format(date));

} catch(ParseException e) {
    System.err.println(" Kein gueltiges Datum (TT.MM.JJJJ)");
}
```

Eine Beschreibung der vordefinierten `DateFormat`-Formate sowie der Definition eigener Formate mit `SimpleDateFormat` finden Sie in Rezept 41. Ein Beispiel für das Einlesen von Datumswerten mit eigenen `SimpleDateFormaten` finden Sie im Start-Programm zu diesem Rezept.

Achtung

Beim Parsen spielt es grundsätzlich keine Rolle, wie oft ein bedeutungstragender Buchstabe, etwa das M für Monatsangaben, wiederholt wird. Während »MM« bei der Formatierung eine zweistellige Ausgabe erzwingt, sind für den Parser »M« und »MM« gleich, d.h., er liest die Anzahl der Monate – egal aus wie vielen Ziffern die Angabe besteht. (Tatsächlich können sogar Werte wie 35 oder 123 übergeben werden. Der überzählige Betrag wird in die nächsthöhere Einheit, für Monate also Jahre, umgerechnet. Wenn Sie dieses Verhalten unterbinden wollen, rufen Sie `setLenient(false)` auf.)

Eine Ausnahme bilden die Jahresangaben. Zweistellige Jahresangaben werden beim Parsen als Kürzel für vierstellige Jahresangaben angesehen und so ergänzt, dass das sich ergebende Datum nicht mehr als 80 Jahre vor und nicht weiter als 20 Jahre hinter dem aktuellen Datum liegt. Angenommen, das Programm wird am 05. April 2005 ausgeführt. Die Eingabe 05.04.24 wird dann als 5. April 2024 geparst. Auch die Eingabe 05.04.25 wird noch ins 21. Jahrhundert verlegt, während die Eingabe 06.04.25 bereits als 6. April 1925 interpretiert wird.

Jahresangaben aus einem oder mehr als zwei Buchstaben (»y«, »yyyy«) werden immer unverändert übernommen.

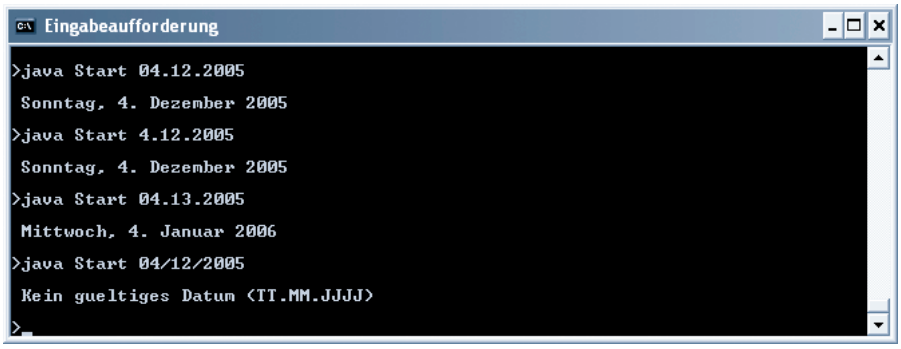


Abbildung 26: Einlesen von Datumseingaben im DateFormat.MEDIUM-Format für die deutsche Lokale. Der dritte Aufruf demonstriert, wie zu große Werte in die nächsthöhere Einheit umgerechnet werden. Der vierte Aufruf zeigt, wie Datumsangaben im angelsächsischen Format abgewiesen werden.

43 Datumswerte vergleichen

Um festzustellen, ob zwei Datumswerte (gegeben als `Date`- oder `Calendar`-Objekt) gleich sind, brauchen Sie nur die Methode `equals()` aufzurufen:

```
// gegeben Date t1 und t2
if (t1.equals(t2))
    System.out.println("gleiche Datumswerte");
```

Um zu prüfen, ob ein Datum zeitlich vor oder nach einem zweiten Datum liegt, stehen Ihnen neben `compareTo()` die speziellen Methoden `before()` und `after()` zur Verfügung:

```
// gegeben Calendar t1 und t2
if (t1.after(t2) )
    System.out.println("\t t1 liegt nach t2");
```

Date-/Calendar-Methode	Beschreibung
`boolean equals(Object)`	Liefert `true`, wenn der aktuelle Datumswert und das übergebene Datum identisch sind. Bei `Date` ist dies der Fall, wenn beide Objekte vom Typ `Date` sind und auf derselben Anzahl Millisekunden basieren. Bei `Calendar` ist dies der Fall, wenn beide Objekte vom Typ `Calendar` sind, auf derselben Anzahl Millisekunden basieren und bestimmte `Calendar`-Charakteristika sowie die Zeitzone übereinstimmen.
`int compareTo(Date/Object)`	Liefert -1, 0 oder 1 zurück, je nachdem, ob der aktuelle Datumswert kleiner, gleich oder größer dem übergebenen Wert ist.
`boolean before(Date/Object)`	Liefert `true`, wenn der aktuelle Datumswert zeitlich vor dem übergebenen Datum liegt.
`boolean after(Date/Object)`	Liefert `true`, wenn der aktuelle Datumswert zeitlich nach dem übergebenen Datum liegt.

Tabelle 19: Vergleichsmethoden für Datumswerte

Vergleiche unter Ausschluss der Uhrzeit

Die vordefinierten Vergleichsmethoden der Klasse `Date` und `Calendar` basieren allesamt auf der Anzahl Millisekunden seit dem 01.01.1970 00:00:00 Uhr, GMT. Sie sind also nicht geeignet, wenn Sie feststellen möchten, ob zwei Datumswerte denselben Tag (ohne Berücksichtigung der Uhrzeit) bezeichnen:

```
// Datumsobjekt für den 5. April 2005, 12 Uhr
Calendar t1 = Calendar.getInstance();
t1.set(2005, 3, 5, 12, 0, 0);

// Datumsobjekt für den 5. April 2005, 13 Uhr
Calendar t2 = Calendar.getInstance();

System.out.println("Vergleich mit equals()   : " + t1.equals(t2));    // false
System.out.println("Vergleich mit compareTo(): " + t1.compareTo(t2)); // -1
```

Um Datumswerte ohne Berücksichtigung der Uhrzeit vergleichen zu können, bedarf es demnach eigener Hilfsmethoden:

```
/**
 *  Prüft, ob zwei Calendar-Objekte den gleichen Tag im Kalender bezeichnen
 */
public static boolean equalDays(Calendar t1, Calendar t2) {
    return   (t1.get(Calendar.YEAR) == t2.get(Calendar.YEAR))
          && (t1.get(Calendar.MONTH) == t2.get(Calendar.MONTH))
          && (t1.get(Calendar.DAY_OF_MONTH) == t2.get(Calendar.DAY_OF_MONTH));
}
```

Die Methode `equalDays()` prüft paarweise, ob Jahr, Monat und Tag der beiden `Calendar`-Objekte übereinstimmen. Wenn ja, liefert sie `true` zurück.

```
/**
 *  Prüft, ob zwei Calendar-Objekte den gleichen Tag bezeichnen
 */
public static int compareDays(Calendar t1, Calendar t2) {
```

```
    Calendar clone1 = (Calendar) t1;
    clone1.set(t1.get(Calendar.YEAR), t1.get(Calendar.MONTH),
               t1.get(Calendar.DATE), 0, 0, 0);
    clone1.clear(Calendar.MILLISECOND);

    Calendar clone2 = (Calendar) t2;
    clone2.set(t2.get(Calendar.YEAR), t2.get(Calendar.MONTH),
               t2.get(Calendar.DATE), 0, 0, 0);
    clone2.clear(Calendar.MILLISECOND);

    return clone1.compareTo(clone2);
}
```


Die Methode `compareDays()` nutzt einen anderen Ansatz als `equalDays()`. Sie legt Kopien der übergebenen `Calendar`-Objekte an und setzt für diese die Werte der Stunden, Minuten, Sekunden (`set()`-Aufruf) sowie Millisekunden (`clear()`-Aufruf) auf 0. Dann vergleicht sie die Klone mit `Calendar.compareTo()` und liefert das Ergebnis zurück.

44 Differenz zwischen zwei Datumswerten berechnen

Wie Sie die Differenz zwischen zwei Datumswerten berechnen, hängt vor allem davon ab, wozu Sie die Differenz benötigen und was Sie daraus ablesen wollen. Geht es lediglich darum, ein Maß für den zeitlichen Abstand zwischen zwei Datumswerten zu erhalten, genügt es, sich die Datumswerte als Anzahl Millisekunden, die seit dem 01.01.1970 00:00:00 Uhr, GMT, vergangen sind, zurückliefern zu lassen und voneinander zu subtrahieren:

```
long diff = Math.abs( date1.getTimeInMillis() - date2.getTimeInMillis() );
```

Hinweis

Wenn Sie mit `Calendar`-Objekten arbeiten, erhalten Sie die Anzahl Millisekunden von der Methode `getTimeInMillis()`. Für `Date`-Objekte rufen Sie stattdessen `getTime()` auf.

Uhrzeit ausschalten

Datumswerte, ob sie nun durch ein Objekt der Klasse `Date` oder `Calendar` repräsentiert werden, schließen immer auch eine Uhrzeit ein. Wenn Sie Datumsdifferenzen ohne Berücksichtigung der Uhrzeit berechnen wollen, müssen Sie die Uhrzeit für alle Datumswerte auf einen gemeinsamen Wert setzen.

Wenn Sie ein neues `GregorianCalendar`-Objekt für ein bestimmtes Datum setzen und nur die Daten für Jahr, Monat und Tag angeben, werden die Uhrzeitfelder automatisch auf 0 gesetzt. Um Ihre Intention deutlicher im Quelltext widerzuspiegeln, können Sie die Felder für Stunden, Minuten und Sekunden aber auch explizit auf 0 setzen:

```
GregorianCalendar date1 = new GregorianCalendar(2002, 5, 1);
GregorianCalendar date2 = new GregorianCalendar(2002, 5, 1,  0, 0, 0);
```

Für bestehende `Calendar`-Objekte können Sie die Uhrzeitfelder mit Hilfe von `set()` oder `clear()` auf 0 setzen:

```
// Aktuelles Datum
Calendar today = Calendar.getInstance();
```

```
// Stunden, Minuten und Sekunden auf 0 setzen
today.set(today.get(Calendar.YEAR), today.get(Calendar.MONTH),
          today.get(Calendar.DATE), 0, 0, 0);
// Millisekunden auf 0 setzen
today.clear(Calendar.MILLISECOND);
```


Eine Beschreibung der verschieden Datums- und Uhrzeitfelder finden Sie in Tabelle 15 aus Rezept 40.

45 Differenz zwischen zwei Datumswerten in Jahren, Tagen und Stunden berechnen

Weit komplizierter ist es, die Differenz zwischen zwei Datumswerten aufgeschlüsselt in Jahre, Tage, Stunden etc. anzugeben. Daran sind vor allem zwei Umstände Schuld:

- Die Sommerzeit.

  Wenn zwei Datumswerte verglichen werden, von denen einer innerhalb und der andere außerhalb der Sommerzeit liegt, führt die Sommerzeitverschiebung zu eventuell unerwünschten Differenzberechnungen.

  In Deutschland beginnt die Sommerzeit am 27. März. Um zwei Uhr nachts wird die Uhr um 1 Stunde vorgestellt. Die Folge: Zwischen dem 27. März 00:00 Uhr und dem 28. März 00:00 Uhr liegen tatsächlich nur 23 Stunden. Trotzdem entspricht dies kalendarisch einem vollen Tag! Wie also sollte ein Programm diese Differenz anzeigen: als 23 h oder als 1 d?

- Die unterschiedlichen Längen der Monate und Jahre.

  Wenn Sie eine Differenz in Jahren und/oder Monaten ausdrücken möchten, stehen Sie vor der Entscheidung, ob Sie mit festen Längen rechnen wollen (1 Jahr = 365 Tage, 1 Monat = 30 Tage oder auch 1 Jahr = 365,25 Tage, 1 Monat = 30,4 Tage) oder ob Sie die exakten Längen berücksichtigen.

Die Klasse `TimeSpan` dient sowohl der Repräsentation als auch der Berechnung von Datumsdifferenzen. In ihren `private`-Feldern speichert sie die Differenz zwischen zwei Datumswerten sowohl in Sekunden (`diff`) als auch ausgedrückt als Kombination aus Jahren, Tagen, Stunden, Minuten und Sekunden.

`TimeSpan`-Objekte können auf zweierlei Weise erzeugt werden:

- indem Sie den `public`-Konstruktor aufrufen und die Differenz selbst als Kombination aus Jahren, Tagen, Stunden, Minuten und Sekunden übergeben:

  ```
  TimeSpan ts = new TimeSpan(0, 1, 2, 0, 0);
  ```

- indem Sie die Methode `getInstance()` aufrufen und dieser zwei `GregorianCalendar`-Objekte übergeben, sowie boolesche Werte, die der Methode mitteilen, ob bei der Berechnung der Differenz auf Sommerzeit und Schaltjahre zu achten ist:

  ```
  GregorianCalendar time1 = new GregorianCalendar(2005, 2, 26);
  GregorianCalendar time2 = new GregorianCalendar(2005, 2, 27);
  TimeSpan ts = TimeSpan.getInstance(time1, time2, true, true);
  ```

Die Differenz, die ein `TimeSpan`-Objekt repräsentiert, können Sie auf zweierlei Weise abfragen:

- Mit Hilfe der get-Methoden (`getYears()`, `getDays()` etc.) lassen Sie sich die Werte der zugehörigen Felder zurückliefern und erhalten so die Kombination aus Jahren, Tagen, Stun-

den, Minuten und Sekunden, aus denen sich die Differenz zusammensetzt. (Die Methode `toString()` liefert auf diese Weise die Differenz als String zurück – nur dass sie natürlich direkt auf die Feldwerte zugreift.)

- Mittels der in-Methoden (`inYears()`, `inWeeks()` etc.) können Sie sich die Differenz ausgedrückt in ganzzahligen Werten einer einzelnen Einheit (also beispielsweise in Jahren oder Tagen) zurückliefern lassen.

Der Quelltext der Klasse `TimeSpan` sieht folgendermaßen aus:

```
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.TimeZone;

/**
 *  Klasse zur Repräsentation und Berechnung von Zeitabständen
 *  zwischen zwei Datumsangaben
 */
public class TimeSpan {
    private int years;
    private int days;
    private int hours;
    private int minutes;
    private int seconds;
    private long diff;

    // public  Konstruktor
    public TimeSpan(int years, int days, int hours,
                    int minutes, int seconds) {
        this.years = years;
        this.days = days;
        this.hours = hours;
        this.minutes = minutes;
        this.seconds = seconds;
        diff = seconds + 60*minutes + 60*60*hours
               + 24*60*60*days + 365*24*60*60*years;
    }

    // protected Konstruktor, wird von getInstance() verwendet
    protected TimeSpan(int years, int days, int hours,
                    int minutes, int seconds, long diff) {
        this.years = years;
        this.days = days;
        this.hours = hours;
        this.minutes = minutes;
        this.seconds = seconds;
        this.diff = diff;
    }

    // Erzeugt aus zwei GregorianCalendar-Objekten
    // ein TimeSpan-Objekt
```

Listing 48: Die Klasse TimeSpan

```
    public static TimeSpan getInstance(GregorianCalendar t1,
                                       GregorianCalendar t2,
                                       boolean summer, boolean leap) {
        // siehe unten
    }

    public int getYears()   { return years; }
    public int getDays()    { return days; }
    public int getHours()   { return hours; }
    public int getMinutes() { return minutes; }
    public int getSeconds() { return seconds; }

    public int inYears()   { return (int) (diff / (60 * 60 * 24 * 365)); }
    public int inWeeks()   { return (int) (diff / (60 * 60 * 24 * 7)); }
    public int inDays()    { return (int) (diff / (60 * 60 * 24)); }
    public int inHours()   { return (int) (diff / (60 * 60)); }
    public int inMinutes() { return (int) (diff / 60); }
    public int inSeconds() { return (int) (diff); }

    public String toString() {
        StringBuilder s = new StringBuilder("");

        if(years > 0)   s.append(years + " j  ");
        if(days > 0)    s.append(days + " t  ");
        if(hours > 0)   s.append(hours + " std  ");
        if(minutes > 0) s.append(minutes + " min  ");
        if(seconds > 0) s.append(seconds + " sec  ");

        if (s.toString().equals("") )
           s.append("Kein Zeitunterschied");

        return s.toString();
    }
}
```

Listing 48: Die Klasse TimeSpan (Forts.)

Am interessantesten ist zweifelsohne die Methode `getInstance()`, die die Differenz zwischen zwei `GregorianCalendar`-Objekten berechnet, indem Sie die Differenz aus den Millisekunden-Werten (zurückgeliefert von `getTimeInMillis()`) berechnet, diesen Wert durch Division mit 1000 in Sekunden umrechnet und dann durch sukzessive Modulo-Berechnung und Division in Sekunden, Minuten, Stunden, Tage und Jahr zerlegt. Am Beispiel der Berechnung des Sekundenanteils möchte ich dies kurz erläutern:

Nachdem die Methode die Differenz durch 1000 dividiert hat, speichert sie den sich ergebenden Wert in der lokalen `long`-Variable `diff`, die somit anfangs die Differenz in Sekunden enthält.

```
diff = (last.getTimeInMillis() - first.getTimeInMillis())/1000;
```

Rechnet man `diff%60` (Modulo = Rest der Division durch 60), erhält man den Sekundenanteil:

```
int seconds = (int) (diff%60);
```

Anschließend wird `diff` durch 60 dividiert und das Ergebnis zurück in `diff` gespeichert.

```
diff /= 60;
```

Jetzt speichert `diff` die Differenz in ganzen Minuten (ohne den Rest Sekunden).


```
public static TimeSpan getInstance(GregorianCalendar t1,
                                   GregorianCalendar t2,
                                   boolean summer, boolean leap) {
    GregorianCalendar first, last;
    TimeZone tz;
    long diff, save;

    // Immer frühere Zeit von späteren Zeit abziehen
    if (t1.getTimeInMillis() > t2.getTimeInMillis()) {
       last = t1;
       first = t2;
    } else {
       last = t2;
       first = t1;
    }

    // Differenz in Sekunden
    diff = (last.getTimeInMillis() - first.getTimeInMillis())/1000;

    if (summer) {            // Sommerzeit ausgleichen
        tz = first.getTimeZone();
        if( !(tz.inDaylightTime(first.getTime()))
              && (tz.inDaylightTime(last.getTime())) )
            diff += tz.getDSTSavings()/1000;
        if(  (tz.inDaylightTime(first.getTime()))
              && !(tz.inDaylightTime(last.getTime())) )
            diff -= tz.getDSTSavings()/1000;
    }

    save = diff;

    // Sekunden, Minuten und Stunden berechnen
    int seconds = (int) (diff%60);  diff /= 60;
    int minutes = (int) (diff%60);  diff /= 60;
    int hours   = (int) (diff%24);  diff /= 24;


    // Jahre und Tage berechnen
    int days = 0;
    int years = 0;

    if (leap) {              // Schaltjahre ausgleichen
       int startYear = 0, endYear = 0;
       int leapDays = 0;               // Schalttage in Zeitraum
       int subtractLeapDays = 0;       // abzuziehende Schalttage
                                       // (da in Jahren enthalten)
```

Listing 49: Quelltext der Methode TimeSpan.getInstance()

```
        if( (first.get(Calendar.MONTH) < 1)
             || (    (first.get(Calendar.MONTH) == 1)
                  && (first.get(Calendar.DAY_OF_MONTH) < 29)))
             startYear = first.get(Calendar.YEAR);
        else
             startYear = first.get(Calendar.YEAR)+1;

        if( (last.get(Calendar.MONTH) > 1)
             || (    (last.get(Calendar.MONTH) == 1)
                  && (last.get(Calendar.DAY_OF_MONTH) == 29)))
             endYear = last.get(Calendar.YEAR);
        else
             endYear = last.get(Calendar.YEAR)-1;

        for(int i = startYear; i <= endYear; ++i)
           if (first.isLeapYear(i))
              ++leapDays;

        // Jahre berechnen
        years   = (int) ((diff-leapDays)/365);

        // in Jahren enthaltene Schalttage
        subtractLeapDays = (years+3)/4;
        if (subtractLeapDays > leapDays)
            subtractLeapDays = leapDays;

        // Tage berechnen
        days    = (int) (diff - ((years*365) + subtractLeapDays));

     } else {
        days    = (int) (diff%365);
        years   = (int) (diff/365);
     }

     return new TimeSpan(years, days, hours, minutes, seconds, (int) save);
  }
```

Listing 49: Quelltext der Methode TimeSpan.getInstance() (Forts.)

Was zum Verständnis der `getInstance()`-Methode noch fehlt, ist die Berücksichtigung von Sommerzeit und Schaltjahren.

Wird für den Parameter `summer` der Wert `true` übergeben, prüft die Methode, ob einer der Datumswerte (aber nicht beide) in die Sommerzeit der aktuellen Zeitzone fallen. Dazu lässt sie sich von der `Calendar`-Methode `getTimeZone()` ein `TimeZone`-Objekt zurückliefern, das die Zeitzone des Kalenders präsentiert, und übergibt nacheinander dessen `inDaylightTime()`-Methode die zu kontrollierenden Datumswerte:

```
    if (summer) {            // Sommerzeit ausgleichen
        tz = first.getTimeZone();
        if( !(tz.inDaylightTime(first.getTime()))
              && (tz.inDaylightTime(last.getTime())) )
```

```
            diff += tz.getDSTSavings()/1000;
        if(  (tz.inDaylightTime(first.getTime()))
              && !(tz.inDaylightTime(last.getTime())) )
            diff -= tz.getDSTSavings()/1000;
    }
```


Fällt tatsächlich einer der Datumswerte in die Sommerzeit und der andere nicht, gleicht die Methode die Sommerzeitverschiebung aus, indem sie sich von der `getDSTSavings()`-Methode des `TimeZone`-Objekts die Verschiebung in Millisekunden zurückliefern lässt und diesen Wert, geteilt durch 1000, auf `diff` hinzuaddiert oder von `diff` abzieht. Die Differenz zwischen dem 27. März 00:00 Uhr und dem 28. März 00:00 Uhr wird dann beispielsweise als 1 Tag und nicht als 23 Stunden berechnet.

Wird für den Parameter `leap` der Wert `true` übergeben, berücksichtigt die Methode in Zeitdifferenzen, die sich über mehrere Jahre erstrecken, Schalttage. Schaltjahre, die in der Differenz komplett enthalten sind, werden demnach als 366 Jahre angerechnet. So wird die Differenz zwischen 01.02.2004 und dem 01.03.2004 zu 29 Tagen berechnet und die Differenz zwischen dem 01.02.2004 und dem 01.02.2005 als genau 1 Jahr. In den meisten Fällen führt diese Berechnung zu Ergebnissen, die man erwartet, sie zeitigt aber auch Merkwürdigkeiten. So werden beispielsweise die Differenzen zwischen dem 28.02.2004 und dem 28.02.2005 zum einen 29.02.2004 und dem 28.02.2005 zum anderen beide zu 1 Jahr berechnet.

Wenn Sie für den Parameter `leap` den Wert `false` übergeben, wird das Jahr immer als 365 Tage aufgefasst.

Das Start-Programm zu diesem Rezept liest über die Befehlszeile zwei deutsche Datumsangaben im Format TT.MM.JJJJ ein und berechnet die Differenz unter Berücksichtigung von Sommerzeit und Schaltjahren.

```
import java.util.GregorianCalendar;
import java.text.DateFormat;
import java.text.ParseException;
import java.util.Locale;

public class Start {

    public static void main(String args[]) {
        DateFormat parser = DateFormat.getDateInstance(DateFormat.MEDIUM,
                                                       Locale.GERMANY);
        GregorianCalendar time1 = new GregorianCalendar();
        GregorianCalendar time2 = new GregorianCalendar();
        TimeSpan ts;
        System.out.println();

        if (args.length != 2) {
           System.out.println(" Aufruf: Start <Datum: TT.MM.JJJJ> "
                              + "<Datum: TT.MM.JJJJ>");
           System.exit(0);
        }

        try {
```

Listing 50: Differenz zwischen zwei Datumswerten berechnen

```
            time1.setTime(parser.parse(args[0]));
            time2.setTime(parser.parse(args[1]));
            ts = TimeSpan.getInstance(time1, time2, true, true);
            System.out.println(" Differenz:  " + ts);

        } catch(ParseException e) {
            System.err.println("\n Kein gueltiges Datum (TT.MM.JJJJ)");
        }
    }
}
```

Listing 50: Differenz zwischen zwei Datumswerten berechnen (Forts.)

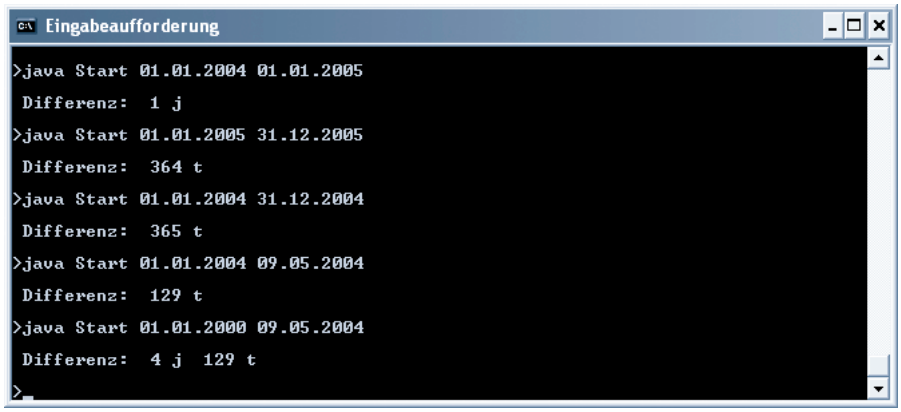


Abbildung 27: Beispielaufrufe

46 Differenz zwischen zwei Datumswerten in Tagen berechnen

Ein Tag besteht stets aus 24 Stunden, 24*60 Minuten, 24*60*60 Sekunden oder 24*60*60*1000 Millisekunden. Was liegt also näher, als die Differenz zwischen zwei Datumswerten zu berechnen, indem man die in den `Calendar`-Objekten gespeicherte Anzahl Millisekunden seit dem 01.01.1970 00:00:00 Uhr, GMT, abfragt, voneinander abzieht, durch 1000 und weiter noch durch 24*60*60 dividiert?

```
// Vereinfachter Ansatz:
long diff = Math.abs((time2.getTimeInMillis()-time1.getTimeInMillis())/1000);
long diffInDays = diff/(60*60*24);
```

Das Problem an dieser Methode ist, dass die Sommerzeit nicht berücksichtigt wird. Stünde in obigem Code `time1` für den 27. März 00:00 Uhr und `time2` für den 28. März 00:00 Uhr, wäre `diff` lediglich gleich 23*60*60 und `diffInDays` ergäbe 0.

Um korrekte Ergebnisse zu erhalten, können Sie entweder die Zeitzone des `Calendar`-Objekts auf eine `TimeZone`-Instanz umstellen, die keine Sommerzeit kennt (und zwar bevor in dem Objekt die gewünschte Zeit gespeichert wird), die Sommerzeitverschiebung manuell korrigieren (siehe Quelltext zu `getInstance()` aus Rezept 45) oder sich der in Rezept 45 definierten `TimeSpan`-Klasse bedienen. In letzterem Fall müssen die beiden Datumswerte als `Gregorian-`

`Calendar`-Objekte vorliegen. Diese übergeben Sie dann an die Methode `getInstance()`, wobei Sie als drittes Argument unbedingt `true` übergeben, damit die Sommerzeit berücksichtigt wird. (Das vierte Argument, das die Berücksichtigung der Schalttage bei der Berechnung der Jahre steuert, ist für die Differenz in Tagen unerheblich.)

```
import java.util.GregorianCalendar;

// gegeben GregorianCalendar time1 und time2
TimeSpan ts = TimeSpan.getInstance(time1, time2, true, true);
System.out.println("Differenz:  " + ts.inDays());
```

Das Start-Programm zu diesem Rezept berechnet auf diese Weise die Differenz in Tagen zwischen zwei Datumseingaben, die über die Befehlszeile entgegengenommen werden.

Achtung

Die Realität ist leider oftmals komplizierter, als wir Programmierer es uns wünschen. Tatsächlich ist in UTC (Coordinated Universal Time) nicht jeder Tag 24*60*60 Sekunden lang. Alle ein oder zwei Jahre wird am Ende des 31. Dezember oder 30 Juni eine Schaltsekunde eingefügt, so dass der Tag 24*60*60+1 Sekunden lang ist. Diese Korrektur gleicht die auf einer Atomuhr basierende UTC-Zeit an die UT-Zeit (GMT) an, die auf der Erdumdrehung beruht.

47 Tage zu einem Datum addieren/subtrahieren

In Sprachen, die die Überladung von Operatoren unterstützen, erwarten Programmieranfänger häufig, dass man das Datum, welches in einem Objekt einer Datumsklasse gekapselt ist, mit Hilfe überladener Operatoren inkrementieren oder um eine bestimmte Zahl Tage erhöhen kann:

```
++date;              // kein Java!
date = date + 3;     // kein Java!
```

Nicht selten sehen sich die Adepten dann getäuscht, weil die zugrunde liegende Implementierung nicht die Anzahl Tage, sondern die Anzahl Millisekunden, auf denen das Datum basiert, erhöhen.

Nun, in Java gibt es keine überladenen Operatoren und obiger Fallstrick bleibt uns erspart. Wie aber kann man in Java Tage zu einem bestehenden Datum hinzuaddieren oder davon abziehen?

Wie Sie mittlerweile wissen, werden Datumswerte in `Calendar`-Objekten sowohl als Anzahl Millisekunden als auch in Form von Datums- und Uhrzeitfeldern (Jahr, Monat, Wochentag, Stunde etc.) gespeichert. Diese Felder können mit Hilfe der get-/set-Methoden der Klasse (siehe Tabelle 15) abgefragt und gesetzt werden.

Eine Möglichkeit, Tage zu einem Datum zu addieren oder von einem Datum abzuziehen, ist daher, `set()` für das Feld `Calendar.DAY_OF_MONTH` aufzurufen und diesem den alten Wert (= `get(Calendar.DAY_OF_MONTH)`) plus der zu addierenden Anzahl Tage (negativer Wert für Subtraktion) zu übergeben.

```
date.set(Calendar.DAY_OF_MONTH, date.get(Calendar.DAY_OF_MONTH) + days);
```

Einfacher noch geht es mit Hilfe der `add()`-Methode, der Sie nur noch das Feld und die zu addierende Anzahl Tage (negativer Wert für Subtraktion) übergeben müssen:

```
date.add(Calendar.DAY_OF_MONTH, days);
```

Kommt es zu einem Über- oder Unterlauf (die berechnete Anzahl Tage ist größer als die Tage im aktuellen Monat bzw. kleiner als 1), passt `add()` die nächstgrößere Einheit an (für Tage also das `MONTH`-Feld). Die `set()`-Methode passt die nächsthöhere Einheit nur dann an, wenn das `private`-Feld `lenient` auf `true` steht (Standardwert, Einstellung über `setLenient()`). Ansonsten wird eine Exception ausgelöst.

Mit der `roll()`-Methode schließlich können Sie den Wert eines Feldes ändern, ohne dass bei Über- oder Unterlauf das nächsthöhere Feld angepasst wird.

```
date.roll(Calendar.DAY_OF_MONTH, days);
```

Methode	Arbeitsweise
`set(int field, int value)`	Anpassung der übergeordneten Einheit, wenn lenient = true, ansonsten Auslösen einer Exception
`add(int field, int value)`	Anpassung der übergeordneten Einheit
`roll(int field, int value)`	Keine Anpassung der übergeordneten Einheit

Tabelle 20: Methoden zum Erhöhen bzw. Vermindern von Datumsfeldern

Das Start-Programm demonstriert die Arbeit von `add()` und `roll()`. Datum und die hinzuzuaddierende Anzahl Tage werden als Argumente über die Befehlszeile übergeben.

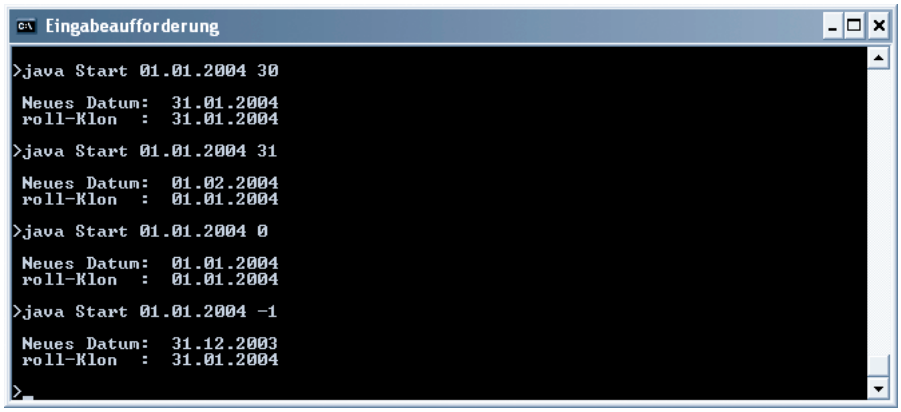


Abbildung 28: Addieren und Subtrahieren von Tagen

48 Datum in Julianischem Kalender

Sie benötigen ein `Calendar`-Objekt, welches den 27.04.2005 im Julianischen Kalender repräsentiert?

In diesem Fall reicht es nicht, einfach dem `GregorianCalendar`-Konstruktor Jahr, Monat (-1) und Tag zu übergeben, da die Hybridimplementierung der Klasse `GregorianCalendar` standardmäßig Daten nach dem 15. Oktober 1582 als Daten im Gregorianischen Kalender interpretiert (vergleiche Rezept 40).

Stattdessen müssen Sie

1. ein neues `GregorianCalendar`-Objekt erzeugen:

   ```
   GregorianCalendar jul = new GregorianCalendar();
   ```

2. dessen `GregorianChange`-Datum auf `Date(Long.MAX_VALUE)` einstellen:

```
jul.setGregorianChange(new Date(Long.MAX_VALUE));
```

3. Jahr, Monat und Tag für das Objekt setzen:

```
jul.set(2005, 3, 27);
```

Das neue Objekt repräsentiert nun das gewünschte Datum im Julianischen Kalender. (Der intern berechnete Millisekundenwert gibt also an, wie viele Sekunden das Datum vom 01.01.1970 00:00:00 Uhr, GMT, entfernt liegt.)

Achtung

Wenn Sie das Datum mittels einer `DateFormat`-Instanz in einen String umwandeln möchten, müssen Sie beachten, dass die `DateFormat`-Instanz standardmäßig mit einer `GregorianCalendar`-Instanz arbeitet, die für Datumswerte nach Oktober 1582 mit dem Gregorianischen Kalender arbeitet. Um das korrekte Julianische Datum zu erhalten, müssen Sie dem Formatierer eine `Calendar`-Instanz zuweisen, die für alle Datumswerte nach dem Julianischen Kalender rechnet, beispielsweise also `jul`:

```
DateFormat dfJul = DateFormat.getDateInstance(DateFormat.FULL);
dfJul.setCalendar(jul);
System.out.println(dfJul.format(jul.getTime()));
```

49 Umrechnen zwischen Julianischem und Gregorianischem Kalender

Um ein Datum im Julianischen Kalender in das zugehörige Datum im Gregorianischen Kalender umzuwandeln (so dass beide Daten gleich viele Millisekunden vom 01.01.1970 00:00:00 Uhr, GMT, entfernt liegen), gehen Sie am besten wie folgt vor:

1. Erzeugen Sie ein neues `GregorianCalendar`-Objekt:

```
GregorianCalendar gc = new GregorianCalendar();
```

2. Stellen Sie dessen `GregorianChange`-Datum auf `Date(Long.MIN_VALUE)` ein:

```
gc.setGregorianChange(new Date(Long.MAX_VALUE));
```

3. Setzen sie die interne Millisekundenzeit des Objekts auf die Anzahl Millisekunden des Julianischen Datums:

```
gc.setTimeInMillis(c.getTimeInMillis());
```

Wenn Sie ein Datum im Gregorianischen Kalender in das zugehörige Datum im Julianischen Kalender umwandeln möchten, gehen Sie analog vor, nur dass Sie `setGregorianChange()` den `Date(Long.MAX_VALUE)` Wert übergeben.

```
/**
 *  Gregorianisches Datum in julianisches Datum unwandeln
 */
public static GregorianCalendar gregorianToJulian(GregorianCalendar c) {

    GregorianCalendar gc = new GregorianCalendar();
    gc.setGregorianChange(new Date(Long.MAX_VALUE));
```

Listing 51: Methoden zur Umwandlung von Datumswerten zwischen Julianischem und Gregorianischem Kalender

```
    gc.setTimeInMillis(c.getTimeInMillis());

    return gc;
}

/**
 *  Julianisches Datum in gregorianisches Datum unwandeln
 */
public static GregorianCalendar julianToGregorian(GregorianCalendar c) {

    GregorianCalendar gc = new GregorianCalendar();
    gc.setGregorianChange(new Date(Long.MIN_VALUE));
    gc.setTimeInMillis(c.getTimeInMillis());

    return gc;
}
```

Listing 51: Methoden zur Umwandlung von Datumswerten zwischen Julianischem und Gregorianischem Kalender (Forts.)

Achtung

Wenn Sie Datumswerte mittels einer `DateFormat`-Instanz in einen String umwandeln:

```
GregorianCalendar date = new GregorianCalendar();
DateFormat df = DateFormat.getDateInstance();
String s = df.format(date.getTime());
```

müssen Sie beachten, dass die `DateFormat`-Instanz das übergebene Datum als `Date`-Objekt übernimmt und mittels einer eigenen `Calendar`-Instanz in Jahr, Monat etc. umrechnet. Wenn Sie mit `DateFormat` Datumswerte umwandeln, für die Sie das `GregorianChange`-Datum umgestellt haben, müssen Sie daher auch für das `Calendar`-Objekt der `DateFormat-Instanz` das `GregorianChange`-Datum umstellen – oder es einfach durch das `GregorianCalendar`-Objekt des Datums ersetzen:

```
GregorianCalendar jul = new GregorianCalendar();
jul.setGregorianChange(new Date(Long.MAX_VALUE));
DateFormat dfJul = DateFormat.getDateInstance(DateFormat.FULL);
dfJul.setCalendar(jul);
```

Das Start-Programm zu diesem Rezept liest ein Datum über die Befehlszeile ein und interpretiert es einmal als Gregorianisches und einmal als Julianisches Datum, welche jeweils in ihre Julianische bzw. Gregorianische Entsprechung umgerechnet werden.

50 Ostersonntag berechnen

Der Ostersonntag ist der Tag, an dem die Christen die Auferstehung Jesu Christi feiern. Gleichzeitig kennzeichnet er das Ende des österlichen Festkreises, der mit dem Aschermittwoch beginnt.

Für den Programmierer ist der Ostersonntag insofern von zentraler Bedeutung, als er den Referenzpunkt für die Berechnung der österlichen Feiertage darstellt: Aschermittwoch, Gründonnerstag, Karfreitag, Ostermontag, Christi Himmelfahrt, Pfingsten.

Der Ostertermin richtet sich nach dem jüdischen Pessachfest und wurde auf dem Konzil von Nicäa 325 festgelegt als:

> »Der 1. Sonntag, der dem ersten Pessach-Vollmond folgt.« – was auf der Nördlichen Halbkugel dem ersten Vollmond nach der Frühlings-Tag-und-Nachtgleiche entspricht.

Wegen dieses Bezugs auf den Vollmond ist die Berechnung des Ostersonntags recht kompliziert. Traditionell erfolgte die Berechnung mit Hilfe des Mondkalenders und der goldenen Zahl (die laufende Nummer eines Jahres im Mondzyklus). Heute gibt es eine Vielzahl von Algorithmen zur Berechnung des Ostersonntags. Die bekanntesten sind die Algorithmen von Carl Friedrich Gauß, Mallen und Oudin. Auf Letzterem basiert auch der in diesem Rezept implementierte Algorithmus:

```
import java.util.GregorianCalendar;

/**
 *  Datum des Ostersonntags im Gregorianischen Kalender berechnen
 */
public static GregorianCalendar eastern(int year) {
    int c = year/100;
    int n = year - 19 * (year/19);
    int k = (c - 17)/25;

    int l1 = c - c/4 - (c-k)/3 + 19*n + 15;
    int l2 = l1 - 30*(l1/30);
    int l3 = l2 - (l2/28)*(1 - (l2/28) * (29/(l2+1)) * ((21-n)/11));

    int a1 = year + year/4 + l3 + 2 - c + c/4;
    int a2 = a1 - 7 * (a1/7);
    int l = l3 - a2;

    int month = 3 + (l + 40)/44;
    int day = l + 28 - 31*(month/4);

    return new GregorianCalendar(year, month-1, day);
}
```

Listing 52: Berechnung des Ostersonntags im Gregorianischen Kalender

Achtung

Wenn Sie historische Ostertermine berechnen, müssen Sie bedenken, dass der Gregorianische Kalender erst im Oktober 1582, in vielen Ländern sogar noch später, siehe Tabelle 16 in Rezept 41, eingeführt wurde.

Wenn Sie zukünftige Ostertermine berechnen, müssen Sie bedenken, dass diese nur nach geltender Konvention gültig sind. Bestrebungen, die Berechnung des Ostertermins zu vereinfachen und das Datum auf einen bestimmten Sonntag festzuschreiben, gibt es schon seit längerem. Bisher konnten die Kirchen diesbezüglich allerdings zu keiner Einigung kommen.

Ostern in der orthodoxen Kirche

Vor der Einführung des Gregorianischen Kalenders galt der Julianische Kalender, nach dem folglich auch Ostern berechnet wurde. Die meisten Länder stellten mit der Übernahme des Gregorianischen Kalenders auch die Berechnung des Ostersonntags auf den Gregorianischen Kalender um. Nicht so die orthodoxen Kirchen. Sie hingen nicht nur lange dem Julianischen Kalender an, siehe Tabelle 16 in Rezept 41, sondern behielten diesen für die Berechnung des Ostersonntags sogar noch bis heute bei.

Die folgende Methode berechnet den Ostersonntag nach dem Julianischen Kalender.

```
import java.util.Date;
import java.util.GregorianCalendar;

/**
 *  Datum des Ostersonntags im Julianischen Kalender berechnen
 */
public static GregorianCalendar easternJulian(int year) {
    int month, day;
    int a = year%19;
    int b = year%4;
    int c = year%7;

    int d = (19 * a + 15) % 30;
    int e = (2*b + 4*c + 6*d + 6)%7;

    if ((d+e) < 10) {
       month = 3;
       day = 22+d+e;
    } else {
       month = 4;
       day = d+e-9;
    }

    GregorianCalendar gc = new GregorianCalendar();
    gc.setGregorianChange(new Date(Long.MAX_VALUE));
    gc.set(year, month-1, day, 0, 0, 0);

    return gc;
}
```

Listing 53: Berechnung des Ostersonntags im Julianischen Kalender

Beachten Sie, dass `GregorianChange`-Datum für das zurückgelieferte `Calendar`-Objekt auf `Date(Long.MAX_VALUE)` gesetzt wurde, d.h., das `Calendar`-Objekt berechnet den Millisekundenwert, der dem übergebenen Datum entspricht, nach dem Julianischen Kalender (siehe auch Rezept 48).

Wenn Sie Jahr, Monat und Tag des Ostersonntags im Julianischen Kalender aus dem zurückgelieferten `Calendar`-Objekt auslesen möchten, brauchen Sie daher nur die entsprechenden Felder abzufragen, beispielsweise:

```
GregorianCalendar easternJ = MoreDate.easternJulian(year);
System.out.println(" " + easternJ.get(Calendar.YEAR)
                 + " " + easternJ.get(Calendar.MONTH)
                 + " " + easternJ.get(Calendar.DAY_OF_MONTH));
```

Wenn Sie das Datum des Ostersonntags im Julianischen Kalender mittels einer `DateFormat`-Instanz in einen String umwandeln möchten, müssen Sie beachten, dass die `DateFormat`-Instanz standardmäßig mit einer `GregorianCalendar`-Instanz arbeitet, die für Datumswerte nach Oktober 1582 den Gregorianischen Kalender zugrunde legt. Um das korrekte Julianische Datum zu erhalten, müssen Sie dem Formatierer eine `Calendar`-Instanz zuweisen, die für alle Datumswerte nach dem Julianischen Kalender rechnet, beispielsweise also das von `MoreDate.easternJulian()` zurückgelieferte Objekt:

```
easternJ = MoreDate.easternJulian(year);
df.setCalendar(easternJ);
System.out.println("Orthod. Ostersonntag (Julian.):  "
                 + df.format(easternJ.getTime()));
```

Sicherlich wird es Sie aber auch interessieren, welchem Datum in unserem Kalender der orthodoxe Ostersonntag entspricht.

Dazu brauchen Sie `easternJ` nur mittels einer `DateFormat`-Instanz zu formatieren, deren `Calendar`-Objekt nicht umgestellt wurde:

```
easternJ = MoreDate.easternJulian(year);
System.out.println("Orthod. Ostersonntag (Gregor.):  "
                 + df.format(easternJ.getTime()));
```

Oder Sie erzeugen eine neue `GregorianCalendar`-Instanz und weisen dieser die Anzahl Millisekunden von `easternJ` zu. Dann können Sie das Datum auch durch Abfragen der Datumsfelder auslesen:

```
GregorianCalendar gc = new GregorianCalendar();
gc.setTimeInMillis(easternJ.getTimeInMillis());
System.out.println(" " + gc.get(Calendar.YEAR) + " " + gc.get(Calendar.MONTH)
                 + " " + gc.get(Calendar.DAY_OF_MONTH));
```

Das Start-Programm zu diesem Rezept demonstriert die Verwendung von `MoreDate.eastern()` und `MoreDate.easternJulian()`. Das Programm nimmt über die Befehlszeile eine Jahreszahl entgegen und gibt dazu das Datum des Ostersonntags aus.

```
import java.util.Date;
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.text.DateFormat;

public class Start {

    public static void main(String args[]) {
        GregorianCalendar eastern, easternJ;
        DateFormat df = DateFormat.getDateInstance(DateFormat.FULL);
        int year = 0;
        System.out.println();
```

Listing 54: Berechnung des Ostersonntags

```
        if (args.length != 1) {
           System.out.println(" Aufruf: Start <Jahreszahl>");
           System.exit(0);
        }

        try {
            year = Integer.parseInt(args[0]);

            // Ostersonntag berechnen
            eastern = MoreDate.eastern(year);
            System.out.println("        Ostersonntag:            "
                               + df.format(eastern.getTime()));


            // Griech-orthodoxen Ostersonntag berechnen
            easternJ = MoreDate.easternJulian(year);
            df.setCalendar(easternJ);
            System.out.println("Orthod. Ostersonntag (Julian.):  "
                               + df.format(easternJ.getTime()));
            df.setCalendar(eastern);
            System.out.println("Orthod. Ostersonntag (Gregor.):  "
                               + df.format(easternJ.getTime()));
        }
        catch (NumberFormatException e) {
            System.err.println(" Ungueltiges Argument");
        }
    }
}
```

Listing 54: Berechnung des Ostersonntags (Forts.)

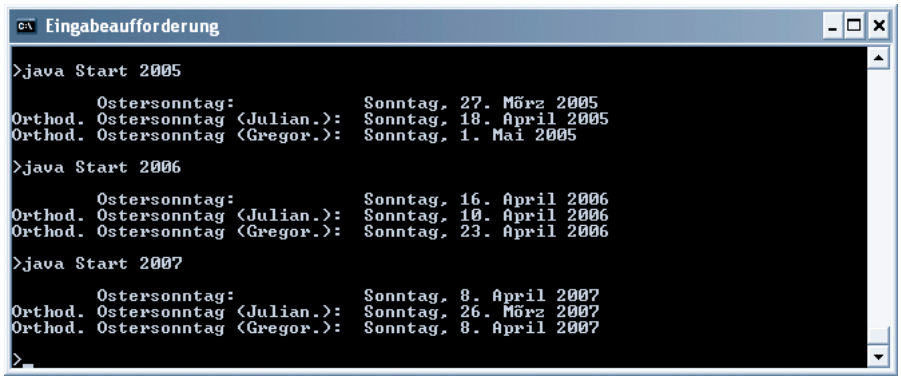


Abbildung 29: Ostersonntage der Jahre 2005, 2006 und 2007

51 Deutsche Feiertage berechnen

Gäbe es nur feste Feiertage, wäre deren Berechnung ganz einfach – ja, eigentlich gäbe es gar nichts mehr zu berechnen, denn Sie müssten lediglich für jeden Feiertag ein `GregorianCalendar`-Objekt erzeugen und dem Konstruktor Jahr, Monat (0-11) und Tag des Feiertagsdatum übergeben.

Fakt ist aber, dass ungefähr die Hälfte aller Feiertage beweglich sind. Da wären zum einen die große Gruppe der Feiertage, die von Osten abhängen, dann die Gruppe der Feiertage, die von Weihnachten abhängen, und schließlich noch der Muttertag.

Letzterer ist im Übrigen kein echter Feiertag, aber wir wollen in diesem Rezept auch die Tage berücksichtigen, denen eine besondere Bedeutung zukommt, auch wenn es sich nicht um gesetzliche Feiertage handelt.

Feiertag	abhängig von	Datum
Neujahr	–	01. Januar
Heilige drei Könige*	–	06. Januar
Rosenmontag	Ostersonntag	Ostersonntag – 48 Tage
Fastnacht	Ostersonntag	Ostersonntag – 47 Tage
Aschermittwoch	Ostersonntag	Ostersonntag – 46 Tage
Valentinstag	–	14. Februar
Gründonnerstag	Ostersonntag	Ostersonntag – 3 Tage
Karfreitag	Ostersonntag	Ostersonntag – 2 Tage
Ostersonntag	Pessach-Vollmond	1. Sonntag, der dem ersten Pessach-Vollmond folgt (siehe Rezept 50)
Ostermontag	Ostersonntag	Ostersonntag + 1 Tag
Maifeiertag	–	1. Mai
Himmelfahrt	Ostersonntag	Ostersonntag + 39 Tage
Muttertag	1. Mai	2. Sonntag im Mai
Pfingstsonntag	Ostersonntag	Ostersonntag + 49 Tage
Pfingstmontag	Ostersonntag	Ostersonntag + 50 Tage
Fronleichnam*	Ostersonntag	Ostersonntag + 60 Tage
Mariä Himmelfahrt*	–	15. September
Tag der deutschen Einheit	–	3. Oktober
Reformationstag*	–	31. Oktober
Allerheiligen*	–	1. November
Allerseelen	–	2. November
Nikolaus	–	6. Dezember
Sankt Martinstag	–	11. November
Volkstrauertag	Heiligabend	Sonntag vor Totensonntag
Buß- und Bettag*	Heiligabend	Mittwoch vor Totensonntag
Totensonntag	Heiligabend	7 Tage vor 1. Advent
1. Advent	Heiligabend	7 Tage vor 2. Advent
2. Advent	Heiligabend	7 Tage vor 3. Advent
3. Advent	Heiligabend	7 Tage vor 4. Advent

*Tabelle 21: Deutsche Feiertage (gesetzliche Feiertage sind farbig hervorgehoben, regionale Feiertage sind mit * gekennzeichnet)*

Feiertag	abhängig von	Datum
4. Advent	Heiligabend	Sonntag vor Heiligabend
Heiligabend	–	24. Dezember
1. Weihnachtstag	–	25. Dezember
2. Weihnachtstag	–	26. Dezember
Silvester	–	31. Dezember

*Tabelle 21: Deutsche Feiertage (gesetzliche Feiertage sind farbig hervorgehoben, regionale Feiertage sind mit * gekennzeichnet) (Forts.)*

Wie Sie der Tabelle entnehmen können, bereitet die Berechnung der Osterfeiertage, insbesondere die Berechnung des Ostersonntags, die größte Schwierigkeit. Doch glücklicherweise haben wir dieses Problem bereits im Rezept 50 gelöst. Die Berechnung der Feiertage reduziert sich damit weitgehend auf die Erzeugung und Verwaltung der Feiertagsdaten. Der hier präsentierte Ansatz basiert auf zwei Klassen:

- einer Klasse `CalendarDay`, deren Objekte die einzelnen Feiertage repräsentieren, und
- einer Klasse `Holidays`, die für ein gegebenes Jahr alle Feiertage berechnet und in einer `Vector`-Collection speichert.

Die Klasse CalendarDay

Die Klasse `CalendarDay` speichert zu jedem Feiertag den Namen, das Datum (als Anzahl Millisekunden), einen optionalen Kommentar, ob es sich um einen gesetzlichen nationalen Feiertag handelt oder ob es ein regionaler Feiertag ist.

```
/**
 *  Klasse zum Speichern von Kalenderinformationen zu Kalendertagen
 */
public class CalendarDay {

    private String name;
    private long time;
    private boolean holiday;
    private boolean nationwide;
    private String comment;

    public CalendarDay(String name, long time, boolean holiday,
                       boolean nationwide, String comment) {
        this.name = name;
        this.time= time;
        this.holiday = holiday;
        this.nationwide = nationwide;
        this.comment = comment;
    }

    public String getName() {
        return name;
```

Listing 55: Die Klasse CalendarDay

```
    }

    public long getTime() {
        return time;
    }

    public boolean getHoliday() {
        return holiday;
    }

    public boolean getNationwide() {
        return nationwide;
    }

    public String getComment() {
        return comment;
    }
}
```

Listing 55: Die Klasse CalendarDay (Forts.)

Die Klasse Holidays

Die Klasse berechnet und verwaltet die Feiertage eines gegebenen Jahres.

Das Jahr übergeben Sie als int-Wert dem Konstruktor, der daraufhin berechnet, auf welche Datumswerte die Feiertage fallen, und für jeden Feiertag ein `CalendarDay`-Objekt erzeugt. Die `CalendarDay`-Objekte werden zusammen in einer `Vector`-Collection gespeichert.

Hinweis

Die Methode `eastern()`, die vom Konstruktor zur Berechnung des Ostersonntags verwendet wird, ist in `Holidays` definiert und identisch zu der Methode aus Rezept 50.

```
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.Vector;

/**
 *  Berechnet Feiertage eines Jahres und speichert die gewonnenen
 *  Informationen in einer Vector-Collection von CalendarDay-Objekten
 */
public class Holidays {
    Vector<CalendarDay> days = new Vector<CalendarDay>(34);

    public Holidays(int year) {
        // Ostern vorab berechnen
        GregorianCalendar eastern = eastern(year);
        GregorianCalendar tmp;
```

Listing 56: Aus Holidays.java

```
int day;

days.add(new CalendarDay("Neujahr",
         (new GregorianCalendar(year,0,1)).getTimeInMillis(),
         true, true, ""));
days.add(new CalendarDay("Heilige drei Könige",
         (new GregorianCalendar(year,0,6)).getTimeInMillis(),
         false, false, "in Baden-Wuert., Bayern und Sachsen-A."));
tmp = (GregorianCalendar) eastern.clone();
tmp.add(Calendar.DAY_OF_MONTH, -48);
days.add(new CalendarDay("Rosenmontag",
         tmp.getTimeInMillis(),
         false, false, ""));
tmp.add(Calendar.DAY_OF_MONTH, +1);
days.add(new CalendarDay("Fastnacht",
         tmp.getTimeInMillis(),
         false, false, ""));
tmp.add(Calendar.DAY_OF_MONTH, +1);
days.add(new CalendarDay("Aschermittwoch",
         tmp.getTimeInMillis(),
         false, false, ""));
days.add(new CalendarDay("Valentinstag",
         (new GregorianCalendar(year,1,14)).getTimeInMillis(),
         false, false, ""));
tmp = (GregorianCalendar) eastern.clone();
tmp.add(Calendar.DAY_OF_MONTH, -3);
days.add(new CalendarDay("Gründonnerstag",
         tmp.getTimeInMillis(),
         false, false, ""));
tmp.add(Calendar.DAY_OF_MONTH, +1);
days.add(new CalendarDay("Karfreitag",
         tmp.getTimeInMillis(),
         true, true, ""));
days.add(new CalendarDay("Ostersonntag",
         eastern.getTimeInMillis(),
         true, true, ""));
tmp = (GregorianCalendar) eastern.clone();
tmp.add(Calendar.DAY_OF_MONTH, +1);
days.add(new CalendarDay("Ostermontag",
         tmp.getTimeInMillis(),
         true, true, ""));
days.add(new CalendarDay("Maifeiertag",
         (new GregorianCalendar(year,4,1)).getTimeInMillis(),
         true, true, ""));
tmp = (GregorianCalendar) eastern.clone();
tmp.add(Calendar.DAY_OF_MONTH, +39);
days.add(new CalendarDay("Himmelfahrt",
         tmp.getTimeInMillis(),
         true, true, ""));

// Muttertag = 2. Sonntag in Mai
```

Listing 56: Aus Holidays.java (Forts.)

```
GregorianCalendar firstMay = new GregorianCalendar(year, 4, 1);
day = firstMay.get(Calendar.DAY_OF_WEEK);
if (day == Calendar.SUNDAY)
    day = 1 + 7;
else
    day = 1 + (8-day) + 7;
days.add(new CalendarDay("Muttertag",
         (new GregorianCalendar(year,4,day)).getTimeInMillis(),
         false, false, ""));

tmp = (GregorianCalendar) eastern.clone();
tmp.add(Calendar.DAY_OF_MONTH, +49);
days.add(new CalendarDay("Pfingstsonntag",
         tmp.getTimeInMillis(),
         true, true, ""));
tmp.add(Calendar.DAY_OF_MONTH, +1);
days.add(new CalendarDay("Pfingstmontag",
         tmp.getTimeInMillis(),
         true, true, ""));
tmp.add(Calendar.DAY_OF_MONTH, +10);
days.add(new CalendarDay("Fronleichnam",
         tmp.getTimeInMillis(),
         true, false, "in Baden-Wuert., Bayern, Hessen, NRW, "
                   + "Rheinl.-Pfalz, Saarland, Sachsen (z.T.) "
                   + "und Thueringen (z.T.)"));
days.add(new CalendarDay("Maria Himmelfahrt",
         (new GregorianCalendar(year,7,15)).getTimeInMillis(),
         false, false, "in Saarland und kathol. Gemeinden "
         + "von Bayern"));
days.add(new CalendarDay("Tag der Einheit",
         (new GregorianCalendar(year,9,3)).getTimeInMillis(),
         true, true, ""));
days.add(new CalendarDay("Reformationstag",
         (new GregorianCalendar(year,9,31)).getTimeInMillis(),
         true, false, "in Brandenburg, Meckl.-Vorp., Sachsen, "
                   + "Sachsen-A. und Thueringen"));
days.add(new CalendarDay("Allerheiligen",
         (new GregorianCalendar(year,10,1)).getTimeInMillis(),
         true, false, "in Baden-Wuert., Bayern, NRW, "
                   + "Rheinl.-Pfalz und Saarland"));
days.add(new CalendarDay("Allerseelen",
         (new GregorianCalendar(year,10,2)).getTimeInMillis(),
         false, false, ""));
days.add(new CalendarDay("Martinstag",
         (new GregorianCalendar(year,10,11)).getTimeInMillis(),
         false, false, ""));

// ab hier nicht mehr chronologisch

// 4. Advent = 1. Sonntag vor 1. Weihnachtstag
GregorianCalendar advent = new GregorianCalendar(year, 11, 25);
```

Listing 56: Aus Holidays.java (Forts.)

```
if (advent.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY)
    advent.add(Calendar.DAY_OF_MONTH, -7);
else
    advent.add(Calendar.DAY_OF_MONTH,
                -advent.get(Calendar.DAY_OF_WEEK)+1);
days.add(new CalendarDay("4. Advent",
         advent.getTimeInMillis(),
         false, false, ""));

// 3. Advent = Eine Woche vor 4. Advent
advent.add(Calendar.DAY_OF_MONTH, -7);
days.add(new CalendarDay("3. Advent",
         advent.getTimeInMillis(),
         false, false, ""));
// 2. Advent = Eine Woche vor 3. Advent
advent.add(Calendar.DAY_OF_MONTH, -7);
days.add(new CalendarDay("2. Advent",
         advent.getTimeInMillis(),
         false, false, ""));
// 1. Advent = Eine Woche vor 2. Advent
advent.add(Calendar.DAY_OF_MONTH, -7);
days.add(new CalendarDay("1. Advent",
         advent.getTimeInMillis(),
         false, false, ""));

// Totensonntag = Sonntag vor 1. Advent
tmp = (GregorianCalendar) advent.clone();
tmp.add(Calendar.DAY_OF_MONTH, -7);
days.add(new CalendarDay("Totensonntag",
         tmp.getTimeInMillis(),
         false, false, ""));

// Volkstrauertag = Sonntag vor Totensonntag
tmp.add(Calendar.DAY_OF_MONTH, -7);
days.add(new CalendarDay("Volkstrauertag",
         tmp.getTimeInMillis(),
         false, false, ""));

// Buß- und Bettag = Mittwoch vor Totensonntag
day = tmp.get(Calendar.DAY_OF_WEEK);
if (day == Calendar.WEDNESDAY)
    day = -(4+day);
else
    day = (4-day);
tmp.add(Calendar.DAY_OF_MONTH, day);
days.add(new CalendarDay("Buß- und Bettag",
         tmp.getTimeInMillis(),
         false, false, "Sachsen"));

days.add(new CalendarDay("Nikolaus",
         (new GregorianCalendar(year,11,6)).getTimeInMillis(),
```

Listing 56: Aus Holidays.java (Forts.)

```
                false, false, ""));
        days.add(new CalendarDay("Heiligabend",
                (new GregorianCalendar(year,11,24)).getTimeInMillis(),
                false, false, ""));
        days.add(new CalendarDay("1. Weihnachtstag",
                (new GregorianCalendar(year,11,25)).getTimeInMillis(),
                true, true, ""));
        days.add(new CalendarDay("2. Weihnachtstag",
                (new GregorianCalendar(year,11,25)).getTimeInMillis(),
                true, true, ""));
        days.add(new CalendarDay("Silvester",
                (new GregorianCalendar(year,11,31)).getTimeInMillis(),
                false, false, ""));
    }
...
```

Listing 56: Aus Holidays.java (Forts.)

Damit man mit der Klasse `Holidays` auch vernünftig arbeiten kann, definiert sie verschiedene Methoden, mit denen der Benutzer Informationen über die Feiertage einholen kann:

- `CalendarDay searchDay(String name)`

Sucht zu einem gegebenen Feiertagsnamen (beispielsweise »Allerheiligen« das zugehörige `CalendarDay`-Objekt und liefert es zurück. Alternative Namen werden zum Teil berücksichtigt. Wurde kein passendes `CalendarDay`-Objekt gefunden, liefert die Methode `null` zurück.

- `CalendarDay getDay(GregorianCalendar date)`

Liefert zu einem gegebenen Datum das zugehörige `CalendarDay`-Objekt zurück bzw. `null`, wenn kein passendes Objekt gefunden wurde.

- `boolean isNationalHoliday(GregorianCalendar date)`

Liefert `true` zurück, wenn auf das übergebene Datum ein gesetzlicher, nationaler Feiertag fällt.

- `boolean isRegionalHoliday(GregorianCalendar date)`

Liefert `true` zurück, wenn auf das übergebene Datum ein gesetzlicher, regionaler Feiertag fällt.

```
...
    // Liefert das CalendarDay-Objekt zu einem Feiertag
    public CalendarDay searchDay(String name) {
        // Alternative Namen berücksichtigen
        if (name.equals("Heilige drei Koenige"))
           name = "Heilige drei Könige";
        if (name.equals("Gruendonnerstag"))
           name = "Gründonnerstag";
        if (name.equals("Tag der Arbeit"))
           name = "Maifeiertag";
        if (name.equals("Christi Himmelfahrt"))
           name = "Himmelfahrt";
```

Listing 57: Die Klasse Holidays

```
        if (name.equals("Vatertag"))
           name = "Himmelfahrt";
        if (name.equals("Tag der deutschen Einheit"))
           name = "Tag der Einheit";
        if (name.equals("Sankt Martin"))
           name = "Martinstag";
        if (name.equals("Vierter Advent"))
           name = "4. Advent";
        if (name.equals("Dritter Advent"))
           name = "3. Advent";
        if (name.equals("Zweiter Advent"))
           name = "2. Advent";
        if (name.equals("Erster Advent"))
           name = "1. Advent";
        if (name.equals("Buss- und Bettag"))
           name = "Buß- und Bettag";
        if (name.equals("Bettag"))
           name = "Buß- und Bettag";
        if (name.equals("Weihnachtsabend"))
           name = "Heiligabend";
        if (name.equals("Erster Weihnachtstag"))
           name = "1. Weihnachtstag";
        if (name.equals("zweiter Weihnachtstag"))
           name = "2. Weihnachtstag";

        for(CalendarDay d : days) {
            if (name.equals(d.getName()) )
               return d;
        }

        return null;
    }

    // Liefert das CalendarDay-Objekt zu einem Kalenderdatum
    public CalendarDay getDay(GregorianCalendar date) {
        for(CalendarDay d : days) {
            if( d.getTime() == date.getTimeInMillis() )
               return d;
        }

        return null;
    }

    // Stellt fest, ob das angegebene Datum auf einen gesetzlichen, nationalen
    // Feiertag fällt
    public boolean isNationalHoliday(GregorianCalendar date) {
        for(CalendarDay d : days) {
            if(   d.getTime() == date.getTimeInMillis()
               && d.getHoliday() && d.getNationwide() )
               return true;
        }
```

Listing 57: Die Klasse Holidays (Forts.)

```
        return false;
    }

    // Stellt fest, ob das angegebene Datum auf einen gesetzlichen, regionalen
    // Feiertag fällt
    public boolean isRegionalHoliday(GregorianCalendar date) {
        for(CalendarDay d : days) {
            if(   d.getTime() == date.getTimeInMillis()
               && d.getHoliday() && !d.getNationwide() )
               return true;
        }

        return false;
    }

    public static GregorianCalendar eastern(int year) {
        // siehe Rezept 50
    }
}
```

Listing 57: Die Klasse Holidays (Forts.)

Zu diesem Rezept gibt es zwei Start-Programme. Beide nehmen die Jahreszahl, für die sie ein `Holidays`-Objekt erzeugen, über die Befehlszeile entgegen.

- Mit *Start1* können Sie die Feiertage eines Jahres auf die Konsole ausgeben oder in eine Datei umleiten:

  ```
  java Start1 > Feiertage.txt
  ```

- Mit *Start2* können Sie abfragen, auf welches Datum ein bestimmter Feiertag im übergebenen Jahr fällt. Der Name des Feiertags wird vom Programm abgefragt.

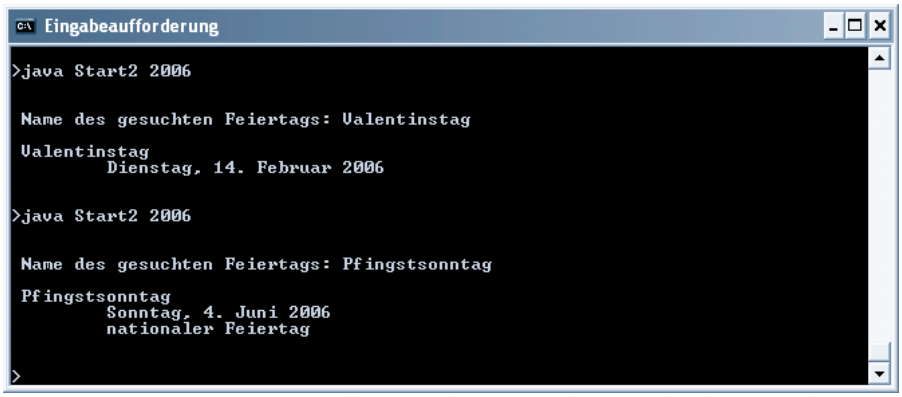


Abbildung 30: Mit Start2 können Sie sich Feiertage in beliebigen Jahren[2] berechnen lassen

2. Immer vorausgesetzt, die entsprechenden Feiertage gibt es in dem betreffenden Jahr und an ihrer Berechnung hat sich nichts geändert. (Denken Sie beispielsweise daran, dass es Bestrebungen gibt, das Osterdatum festzuschreiben.)

52 Ermitteln, welchen Wochentag ein Datum repräsentiert

Welchem Wochentag ein Datum entspricht, ist im `DAY_OF_WEEK`-Feld des `Calendar`-Objekts gespeichert. Für Instanzen von `GregorianCalendar` enthält dieses Feld eine der Konstanten `SUNDAY`, `MONDAY`, `TUESDAY`, `WEDNESDAY`, `THURSDAY`, `FRIDAY` oder `SATURDAY`.

Den Wert des Felds können Sie für ein bestehendes `Calendar`-Objekt `date` wie folgt abfragen:

```
int day = date.get(Calendar.DAY_OF_WEEK);
```

Für die Umwandlung der `DAY_OF_WEEK`-Konstanten in Strings (»Sonntag«, »Montag« etc.) ist es am einfachsten, ein Array der Wochentagsnamen zu definieren und den von `get(Calendar.DAY_OF_WEEK)` zurückgelieferten String als Index in dieses Array zu verwenden. Sie müssen allerdings beachten, dass die `DAY_OF_WEEK`-Konstanten den Zahlen von 1 (`SUNDAY`) bis 7 (`SATURDAY`) entsprechen, während Arrays mit 0 beginnend indiziert werden.

Das Start-Programm zu diesem Rezept, welches den Wochentag zu einem beliebigen Datum ermittelt, demonstriert diese Technik:

```
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.text.DateFormat;
import java.text.ParseException;
import java.util.Locale;

public class Start {

    public static void main(String args[]) {
        DateFormat df = DateFormat.getDateInstance(DateFormat.MEDIUM,
                                                   Locale.GERMANY);
        GregorianCalendar date = new GregorianCalendar();
        int day;
        String[] weekdayNames = { "SONNTAG", "MONTAG", "DIENSTAG", "MITTWOCH",
                                  "DONNERSTAG", "FREITAG", "SAMSTAG" };

        System.out.println();

        if (args.length != 1) {
           System.out.println(" Aufruf: Start <Datum: TT.MM.JJJJ>");
           System.exit(0);
        }

        try {
            date.setTime(df.parse(args[0]));

            day = date.get(Calendar.DAY_OF_WEEK);

            System.out.println(" Wochentag:  " + weekdayNames[day-1]);

        } catch(ParseException e) {
            System.err.println("\n Kein gueltiges Datum (TT.MM.JJJJ)");
```

Listing 58: Programm zur Berechnung des Wochentags

```
        }
    }
}
```

Listing 58: Programm zur Berechnung des Wochentags (Forts.)

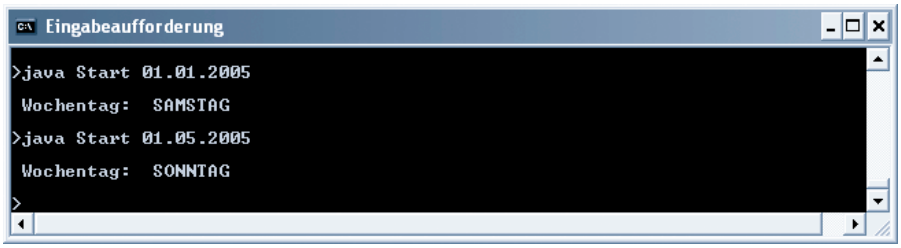


Abbildung 31: Kann ein Maifeiertag schlechter liegen?

53 Ermitteln, ob ein Tag ein Feiertag ist

Mit Hilfe der Klasse `Holidays` aus Rezept 51 können Sie schnell prüfen, ob es sich bei einem bestimmten Tag im Jahr um einen Feiertag handelt.

1. Zuerst erzeugen Sie für das gewünschte Jahr ein `Holidays`-Objekt.

   ```
   Holidays holidays = new Holidays(2005);
   ```

2. Dann erzeugen Sie ein `GregorianCalendar`-Objekt für das zu untersuchende Datum.

   ```
   GregorianCalendar date = new GregorianCalendar(2005, 3, 23);
   ```

3. Schließlich prüfen Sie mit Hilfe der entsprechenden Methoden des `Holidays`-Objekts, ob es sich um einen Feiertag handelt.

   Sie können dabei beispielsweise so vorgehen, dass Sie zuerst durch Aufruf von `isNationalHoliday()` prüfen, ob es sich um einen nationalen gesetzlichen Feiertag handelt. Wenn nicht, können Sie mit `isRegionalHoliday()` prüfen, ob es ein regionaler gesetzlicher Feiertag ist. Trifft auch dies nicht zu, können Sie mit `getDay()` prüfen, ob der Tag überhaupt als besonderer Tag in dem `holidays`-Objekt gespeichert ist:

   ```
   if(holidays.isNationalHoliday(date)) {
       System.out.println("\t Nationaler Feiertag ");
   } else if(holidays.isRegionalHoliday(date)) {
       System.out.println("\t Regionaler Feiertag " );
   } else if (holidays.getDay(date) != null) {
       System.out.println("\t Besonderer Tag " );
   } else
       System.out.println("\t Kein Feiertag ");
   ```

Bezeichnet ein Datum einen Tag, der im `Holidays`-Objekt gespeichert ist, können Sie sich mit `getDay()` die Referenz auf das zugehörige `CalendarDay`-Objekt zurückliefern lassen und die für den Tag gespeicherten Informationen abfragen.

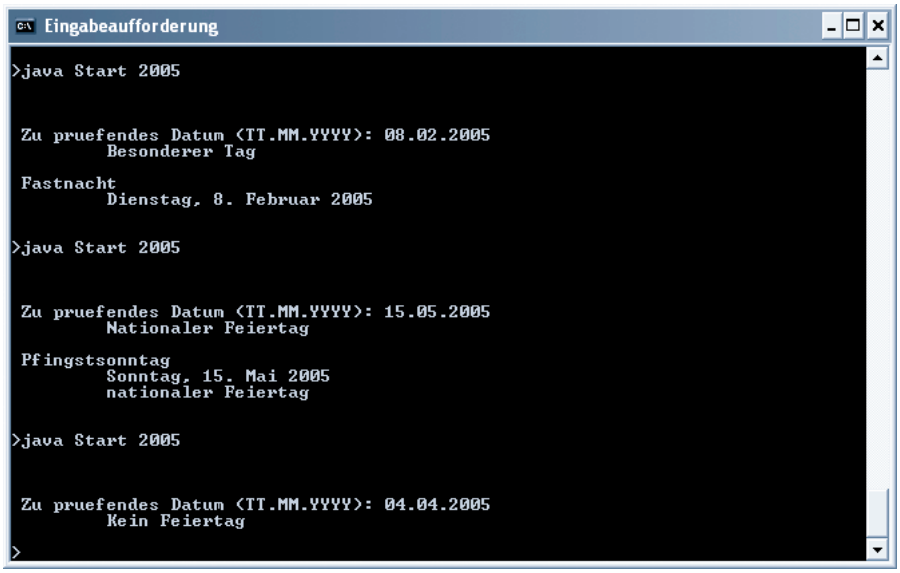


Abbildung 32: Ermitteln, ob ein Tag ein Feiertag ist

54 Ermitteln, ob ein Jahr ein Schaltjahr ist

Ob ein gegebenes Jahr im Gregorianischen Kalender ein Schaltjahr ist, lässt sich bequem mit Hilfe der Methode `isLeapYear()` feststellen. Leider ist die Methode nicht statisch, so dass Sie zum Aufruf ein `GregorianCalendar`-Objekt benötigen. Dieses muss aber nicht das zu prüfende Jahr repräsentieren, die Jahreszahl wird vielmehr als Argument an den `int`-Parameter übergeben.

```
GregorianCalendar date = new GregorianCalendar();
int year = 2005;

date.isLeapYear(year);
```

Mit dem Start-Programm zu diesem Rezept können Sie prüfen, ob ein Jahr im Gregorianischen Kalender ein Schaltjahr ist.

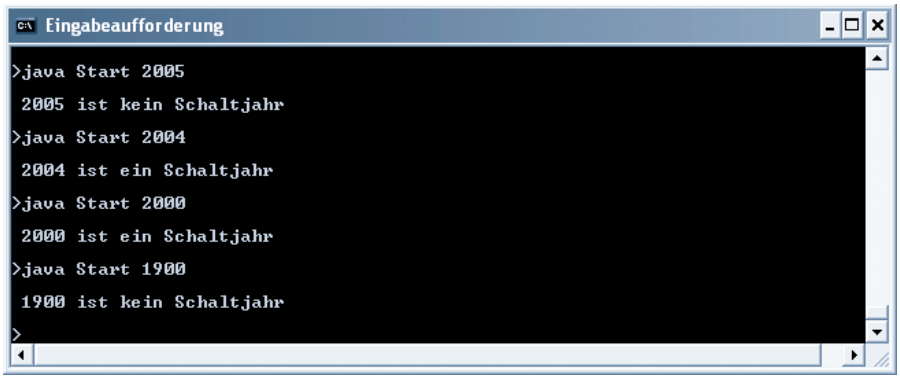


Abbildung 33: 1900 ist kein Schaltjahr, weil es durch 100 teilbar ist. 2000 ist ein Schaltjahr, obwohl es durch 100 teilbar ist, weil es ein Vielfaches von 400 darstellt.

55 Alter aus Geburtsdatum berechnen

Die Berechnung des Alters, sei es nun das Alter eines Kunden, einer Ware oder einer beliebigen Sache (wie z.B. Erfindungen), ist eine recht häufige Aufgabe. Wenn lediglich das Jahr der »Geburt« bekannt ist, ist diese Aufgabe auch relativ schnell durch Differenzbildung der Jahreszahlen erledigt.

Liegt jedoch das komplette Geburtsdatum vor und ist dieses mit einem zweiten Datum, beispielsweise dem aktuellen Datum, zu vergleichen, müssen Sie beachten, dass das Alter unter Umständen um 1 geringer ist als die Differenz der Jahreszahlen – dann nämlich, wenn das Vergleichsdatum in seinem Jahr weiter vorne liegt als das Geburtsdatum im Geburtsjahr. Die Methode `age()` berücksichtigt dies:

```
/**
 *  Berechnet, welches Alter eine Person, die am birthdate
 *  geboren wurde, am otherDate hat
 */
public static int age(Calendar birthdate, Calendar otherDate) {
    int age = 0;

    // anderes Datum liegt vor Geburtsdatum
    if (otherDate.before(birthdate))
        return  -1;

    // Jahresunterschied berechnen
    age = otherDate.get(Calendar.YEAR) - birthdate.get(Calendar.YEAR);

    // Prüfen, ob Tag in otherDate vor Tag in birthdate liegt. Wenn ja,
    // Alter um 1 Jahr vermindern
    if ( (otherDate.get(Calendar.MONTH) < birthdate.get(Calendar.MONTH))
       ||(otherDate.get(Calendar.MONTH) == birthdate.get(Calendar.MONTH)
          && otherDate.get(Calendar.DAY_OF_MONTH) <
             birthdate.get(Calendar.DAY_OF_MONTH)))
       --age;

    return age;
}
```

Vielleicht wundert es Sie, dass die Methode so scheinbar umständlich prüft, ob der Monat im Vergleichsjahr kleiner als der Monat im Geburtsjahr ist, oder, falls die Monate gleich sind, der Tag im Monat des Vergleichsjahrs kleiner dem Tag im Monat des Geburtsjahrs ist. Könnte man nicht einfach das Feld `DAY_OF_YEAR` für beide Daten abfragen und vergleichen?

Die Antwort ist nein, weil dann Schalttage das Ergebnis verfälschen können. Konkret: Für das Geburtsdatum 01.03.1955 und ein Vergleichsdatum 29.02.2004 würde `get(Calendar.DAY_OF_YEAR)` in beiden Fällen 60 zurückliefern. Das berechnete Alter wäre daher fälschlicherweise 50 statt 49.

Mit dem Start-Programm zu diesem Rezept können Sie berechnen, wie alt eine Person oder ein Gegenstand heute ist. Das Geburtsdatum wird im Programmverlauf abgefragt, das Vergleichsdatum ist das aktuelle Datum. Beachten Sie auch die Formatierung der Ausgabe mit `ChoiceFormat`, siehe Rezept 11.

```
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.text.DateFormat;
import java.text.ChoiceFormat;
import java.text.ParseException;
import java.util.Locale;
import java.util.Scanner;

public class Start {

    public static void main(String args[]) {
        DateFormat df = DateFormat.getDateInstance(DateFormat.MEDIUM,
                                                   Locale.GERMANY);
        GregorianCalendar date = new GregorianCalendar();
        Scanner sc = new Scanner(System.in);
        int age = 0;

        try {
            System.out.print("\n Geben Sie Ihr Geburtsdatum im Format "
                             + " TT.MM.JJJJ ein: ");
            String input = sc.next();

            date.setTime(df.parse(input));

            // Vergleich mit aktuellem Datum
            age = MoreDate.age(date, Calendar.getInstance());

            if (age < 0) {
                System.out.println("\n Sie sind noch nicht geboren");
            } else {
                double[] limits =  {0, 1, 2};
                String[] outputs = {"Jahre", "Jahr", "Jahre"};
                ChoiceFormat cf = new ChoiceFormat(limits, outputs);

                System.out.println("\n Sie sind " + age + " "
                                   + cf.format(age) + " alt");
            }

        } catch(ParseException e) {
            System.err.println("\n Kein gueltiges Datum (TT.MM.JJJJ)");
        }
    }
}
```

Listing 59: Start.java – Programm zur Altersberechnung

56 Aktuelle Zeit abfragen

Der einfachste und schnellste Weg, die aktuelle Zeit abzufragen, besteht darin, ein Objekt der Klasse `Date` zu erzeugen:

```
import java.util.Date;

Date today = new Date();
System.out.println(today);
```

Ausgabe:

```
Thu Mar 31 10:54:31 CEST 2005
```

Wenn Sie lediglich die Zeit ausgeben möchten, lassen Sie sich von `DateFormat.getTimeInstance()` ein entsprechendes Formatierer-Objekt zurückliefern und übergeben Sie das `Date`-Objekt dessen `format()`-Methode. Als Ergebnis erhalten Sie einen formatierten Uhrzeit-String zurück.

```
String s = DateFormat.getTimeInstance().format(today);
System.out.println( s );                                    // Ausgabe: 10:54:31
```

Hinweis

Mehr zur Formatierung mit `DateFormat`, siehe Rezept 41.

Sofern Sie die Uhrzeit nicht nur ausgeben oder bestenfalls noch mit anderen Uhrzeiten des gleichen Tags vergleichen möchten, sollten Sie die Uhrzeit durch ein `Calendar`-Objekt (siehe auch Rezept 39) repräsentieren.

- Sie können sich mit `getInstance()` ein `Calendar`-Objekt zurückliefern lassen, welches die aktuelle Zeit (natürlich inklusive Datum) repräsentiert:

  ```
  Calendar calendar = Calendar.getInstance();
  ```

- Sie können ein `Calendar`-Objekt erzeugen und auf eine beliebige Zeit setzen:

  ```
  Calendar calendar = Calendar.getInstance();
  ```

- Sie können die Zeit aus einem `Date`-Objekt an ein `Calendar`-Objekt übergeben:

  ```
  Calendar calendar = Calendar.getInstance();
  calendar.set(calendar.get(Calendar.YEAR),        // Datum
               calendar.get(Calendar.MONTH),       // beibehalten
               calendar.get(Calendar.DATE),
               12, 30, 1);                      // Uhrzeit setzen
  ```

- Sie können ein `GregorianCalender`-Objekt für eine bestimmte Uhrzeit erzeugen:

  ```
  GregorianCalendar gCal =
                                    // year, m, d,   h, min, sec
                  new GregorianCalendar(2005, 4, 20,  12, 30, 1);
  ```

Hinweis

Um die in einem `Calendar`-Objekt gespeicherte Uhrzeit auszugeben, können Sie entweder die Werte für die einzelnen Uhrzeit-Felder mittels der zugehörigen get-Methoden abfragen (siehe Tabelle 15 aus Rezept 40) und in einen String/Stream schreiben oder sie wandeln die Feldwerte durch Aufruf von `getTime()` in ein `Date`-Objekt um und übergeben dieses an die `format()`-Methode einer `DateFormat`-Instanz:

```
String s = DateFormat.getTimeInstance().format(calendar.getTime()) );
```

57 Zeit in bestimmte Zeitzone umrechnen

Wenn Sie mit Hilfe von `Date` oder `Calendar` die aktuelle Zeit abfragen (siehe Rezept 56), wird das Objekt mit der Anzahl Millisekunden initialisiert, die seit dem 01.01.1970 00:00:00 Uhr, GMT, vergangen sind. Wenn Sie diese Zeitangabe in einen formatierten String umwandeln lassen (mittels `DateFormat` oder `SimpleDateFormat`, siehe Rezept 41), wird die Anzahl Millisekunden gemäß dem gültigen Kalender und gemäß der auf dem aktuellen System eingestellten Zeitzone in Datums- und Zeitfelder (Jahr, Monat, Tag, Stunde, Minute etc.) umgerechnet.

Formatierer auf Zeitzone umstellen

Wenn Sie die Zeit dagegen in die Zeit einer anderen Zeitzone umrechnen lassen möchten, gehen Sie wie folgt vor:

1. Erzeugen Sie ein `TimeZone`-Objekt für die gewünschte Zeitzone.
2. Registrieren Sie das `TimeZone`-Objekt beim Formatierer.
3. Wandeln Sie die Zeitangabe mit Hilfe des Formatierers in einen String um.

Um beispielsweise zu berechnen, wie viel Uhr es aktuell in Los Angeles ist, würden Sie schreiben:

```
// Formatierer
DateFormat df = DateFormat.getDateTimeInstance(DateFormat.FULL,
                                               DateFormat.FULL);

// Aktuelles Datum
Date today = new Date();

// 1. Zeitzone erzeugen
TimeZone tz = TimeZone.getTimeZone("America/Los_Angeles");

// 2. Zeitzone beim Formatierer registrieren
df.setTimeZone(tz);

// 3. Umrechnung (und Ausgabe) in Zeitzone für Los Angeles (Amerika)
System.out.println(" America/Los Angeles: " + df.format(today));
```

Hinweis

Wenn die Uhrzeit in Form eines `Calendar`-Objekts vorliegt, gehen Sie analog vor. Sie müssen lediglich daran denken, die Daten des `Calendar`-Objekts als `Date`-Objekt an die `format()`-Methode zu übergeben: `df.format(calObj.getTime())`.

Calendar auf Zeitzone umstellen

Sie können auch das `Calendar`-Objekt selbst auf eine andere Zeitzone umstellen. In diesem Fall übergeben Sie das `TimeZone`-Objekt, welches die Zeitzone repräsentiert, mittels `setTimeZone()` an das `Calendar`-Objekt:

```
Calendar calendar = Calendar.getInstance();
...
TimeZone tz = TimeZone.getTimeZone("America/Los_Angeles");
calendar.setTimeZone(tz);
```

Die get-Methoden des `Calendar`-Objekts – wie z.B. `calendar.get(calendar.HOUR_OF_DAY)`, `calendar.get(calendar.DST_OFFSET)`, siehe Tabelle 15 aus Rezept 40 – liefern daraufhin die der Zeitzone entsprechenden Werte (inklusive Zeitverschiebung und Berücksichtigung der Sommerzeit) zurück.

Achtung

Die Zeit, die ein `Calendar`-Objekt repräsentiert, wird intern als Anzahl Millisekunden, die seit dem 01.01.1970 00:00:00 Uhr, GMT vergangen sind, gespeichert. Dieser Wert wird durch die Umstellung auf eine Zeitzone nicht verändert. Es ändern sich lediglich die Datumsfeldwerte, wie Stunden, Minuten etc., die intern aus der Anzahl Millisekunden unter Berücksichtigung der Zeitzone berechnet werden. Vergessen Sie dies nie, vor allem nicht bei der Formatierung der Zeitwerte mittels `DateFormat`. Wenn Sie sich nämlich mit `getTime()` ein `Date`-Objekt zurückliefern lassen, das Sie der `format()`-Methode von `DateFormat` übergeben können, wird dieses `Date`-Objekt auf der Grundlage der intern gespeicherten Anzahl Millisekunden erzeugt. Soll `DateFormat` die Uhrzeit in der Zeitzone des `Calendar`-Objekts formatieren, müssen Sie die Zeitzone des `Calendar`-Objekts zuvor beim Formatierer registrieren:

```
df.setTimeZone(calendar.getTimeZone());
```

58 Zeitzone erzeugen

Zeitzonen werden in Java durch Objekte vom Typ der Klasse `TimeZone` repräsentiert. Da `TimeZone` selbst abstrakt ist, lassen Sie sich `TimeZone`-Objekte von der statischen Methode `getTimeZone()` zurückliefern, der Sie als Argument den ID-String der gewünschten Zeitzone übergeben:

```
TimeZone tz = TimeZone.getTimeZone("Europe/Berlin");
```

ID	Zeitzone	entspricht GMT
Pacific/Samoa	Samoa Normalzeit	GMT-11:00
US/Hawaii	Hawaii Normalzeit	GMT-10:00
US/Alaska	Alaska Normalzeit	GMT-09:00
US/Pacific, Canada/Pacific, America/Los_Angeles	Pazifische Normalzeit	GMT-08:00
US/Mountain, Canada/Mountain, America/Denver	Rocky Mountains Normalzeit	GMT-07:00
US/Central, America/Chicago, America/ Mexico_City	Zentrale Normalzeit	GMT-06:00

Tabelle 22: Zeitzonen

ID	Zeitzone	entspricht GMT
US/Eastern, Canada/Eastern, America/New_York	Östliche Normalzeit	GMT-05:00
Canada/Atlantic, Atlantic/Bermuda	Atlantik Normalzeit	GMT-04:00
America/Buenos_Aires	Argentinische Zeit	GMT-03:00
Atlantic/South_Georgia	South Georgia Normalzeit	GMT-02:00
Atlantic/Azores	Azoren Zeit	GMT-01:00
Europe/Dublin, Europe/London, Africa/Dakar Etc/UTC	Greenwich Zeit Koordinierte Universalzeit	GMT-00:00
Europe/Berlin, Etc/GMT-1	Zentraleuropäische Zeit	GMT+01:00
Europe/Kiev Africa/Cairo Asia/Jerusalem	Osteuropäische Zeit Zentralafrikanische Zeit Israelische Zeit	GMT+02:00
Europe/Moscow Asia/Baghdad	Moskauer Normalzeit Arabische Normalzeit	GMT+03:00
Asia/Dubai	Golf Normalzeit	GMT+04:00
Indian/Maledives	Maledivische Normalzeit	GMT+05:00
Asia/Colombo	Sri Lanka Zeit	GMT+06:00
Asia/Bangkok	Indochina Zeit	GMT+07:00
Asia/Shanghai	Chinesische Normalzeit	GMT+08:00
Asia/Tokyo	Japanische Normalzeit	GMT+09:00
Australia/Canberra	Östliche Normalzeit	GMT+10:00
Pacific/Guadalcanal	Salomoninseln Zeit	GMT+11:00
Pacific/Majuro	Marshallinseln Zeit	GMT+12:00

Tabelle 22: Zeitzonen (Forts.)

Hinweis

Die weit verbreiteten dreibuchstabigen Zeitzonen-Abkürzungen wie ETC, PST, CET, die aus Gründen der Abwärtskompatibilität noch unterstützt werden, sind nicht eindeutig und sollten daher möglichst nicht mehr verwendet werden.

Verfügbare Zeitzonen abfragen

Die Übergabe einer korrekten ID ist aber noch keine Garantie, dass die zur Erstellung des `TimeZone`-Objekts benötigten Informationen auf dem aktuellen System vorhanden sind. Dazu müssen Sie sich mit `TimeZone.getAvailableIDs()` ein String-Array mit den IDs der auf dem System verfügbaren Zeitzonen zurückliefern lassen und prüfen, ob die gewünschte ID darin vertreten ist.

```
TimeZone tz = null;
String ids[] = TimeZone.getAvailableIDs();
for (int i = 0; i < ids.length; ++i)
```

```
    if (ids[i].equals(searchedID))
        tz = TimeZone.getTimeZone(ids[i]);
```

Wenn Sie `TimeZone.getTimeZone()` eine ungültige ID übergeben, erhalten Sie die Greenwich-Zeitzone (»GMT«) zurück.

Eigene Zeitzonen erzeugen

Eigene Zeitzonen erzeugen Sie am einfachsten, indem Sie `TimeZone.getTimeZone()` als ID einen String der Form »GMT-hh:mm« bzw. »GMT+hh:mm« übergeben, wobei hh:mm die Zeitverschiebung in Stunden und Minuten angibt.

```
TimeZone tz = TimeZone.getTimeZone("GMT-01:00");
```

Allerdings berücksichtigen die erzeugten `TimeZone`-Objekte dann keine Sommerzeit. Dazu müssen Sie nämlich explizit ein Objekt der Klasse `SimpleTimeZone` erzeugen und deren Konstruktor, neben der frei wählbaren ID für die neue Zeitzone, auch noch die Informationen für Beginn und Ende der Sommerzeit übergeben.

Die im Folgenden abgedruckte Methode `MoreDate.getTimeZone()` verfolgt eine zweigleisige Strategie. Zuerst prüft sie, ob die angegebene ID in der Liste der verfügbaren IDs zu finden ist. Wenn ja, erzeugt sie direkt anhand der ID das gewünschte `TimeZone`-Objekt. Bis hierher unterscheidet sich die Methode noch nicht von einem direkten `TimeZone.getTimeZone()`-Aufruf. Sollte die Methode allerdings feststellen, dass es zu der ID keine passenden Zeitzonen-Informationen gibt, liefert sie nicht die GMZ-Zeitzone zurück, sondern zieht die ebenfalls als Argumente übergebenen Informationen zu Zeitverschiebung und Sommerzeit hinzu und erzeugt ein eigenes `SimpleTimeZone`-Objekt.

```
/**
 *  Hilfsmethode zum Erzeugen einer Zeitzone (TimeZone-Objekt)
 */
public static TimeZone getTimeZone(String id, int rawOffset,
                                   int startMonth, int startDay,
                                   int startDayOfWeek, int startTime,
                                   int endMonth, int endDay,
                                   int endDayOfWeek, int endTime,
                                   int dstSavings) {
    TimeZone tz = null;

    // Ist gewünschte Zeitzone verfügbar?
    String ids[] = TimeZone.getAvailableIDs();
    for (int i = 0; i < ids.length; ++i)
        if (ids[i].equals(id))
            tz = TimeZone.getTimeZone(ids[i]);

    if(tz == null)      // Eigene Zeitzone konstruieren
       tz = new SimpleTimeZone(rawOffset, id, startMonth, startDay,
                               startDayOfWeek, startTime, endMonth,
                               endDay, endDayOfWeek, endTime,
                               dstSavings);

    return tz;
}
```

Das Start-Programm zu diesem Rezept zeigt den Aufruf von `MoreDate.getTimeZone()`, um sich ein `TimeZone`-Objekt für »America/Los_Angeles« zurückliefern zu lassen:

```
// aus Start.java
SimpleDateFormat sdf = new SimpleDateFormat("dd. MMMM yyyy, HH:mm");
Calendar calendar = Calendar.getInstance();
TimeZone tz;

tz = MoreDate.getTimeZone("America/Los_Angeles", -28800000,
                          Calendar.APRIL, 1, -Calendar.SUNDAY, 7200000,
                          Calendar.OCTOBER, -1, Calendar.SUNDAY, 7200000,
                          3600000);
sdf.setTimeZone(tz);
System.out.println("\t" + sdf.format(calendar.getTime()));
```

Ausgabe:

```
04. April 2005, 05:10
```

Achtung

Wenn Sie die Uhrzeit mit Angabe der Zeitzonen ausgeben:

```
SimpleDateFormat sdf = new SimpleDateFormat("dd. MMMM yyyy, HH:mm z");
```

kann es passieren, dass für selbst definierte Zeitzonen (ID nicht in der Liste der verfügbaren IDs vorhanden) eine falsche Zeitzone angezeigt wird. Dies liegt daran, dass `SimpleDateFormat` in diesem Fall in die Berechnung der »Zeitzone« auch die Sommerzeitverschiebung mit einbezieht.

59 Differenz zwischen zwei Uhrzeiten berechnen

Die Differenz zwischen zwei Uhrzeiten zu berechnen, ist grundsätzlich recht einfach: Sie lassen sich die beiden Zeiten als Anzahl Millisekunden seit dem 01.01.1970 00:00:00 Uhr, GMT, zurückgeben, bilden durch Subtraktion die Differenz und rechnen das Ergebnis in die gewünschte Einheit um:

```
import java.util.GregorianCalendar;

GregorianCalendar time1 = new GregorianCalendar(2005, 4, 1, 22, 30, 0);
GregorianCalendar time2 = new GregorianCalendar(2005, 4, 2,  7, 30, 0);

long diff = time2.getTimeInMillis() - time1.getTimeInMillis();

// Differenz in Millisekunden : diff
// Differenz in Sekunden      : diff/1000
// Differenz in Minuten       : diff/(60*1000)
// Differenz in Stunden       : diff/(60*60*1000)
```

Das obige Verfahren berechnet letzten Endes aber keine Differenz zwischen Uhrzeiten, sondern Differenzen zwischen Zeiten (inklusive Datum). Das heißt, für `time1` = 01.05.2005 22:30 Uhr und `time2` = 03.05.2005 7:30 Uhr würde die Berechnung 33 Stunden (bzw. 1980 Minuten) ergeben. Dies kann, muss aber nicht im Sinne des Programmierers liegen.

Wenn Sie nach obigem Verfahren den zeitlichen Abstand zwischen zwei reinen Uhrzeiten (beispielsweise von 07:00 zu 14:00 oder von 14:00 zu 05:00 am nächsten Tag) so berechnen wollen, wie man ihn am Zifferblatt einer Uhr ablesen würde, müssen Sie darauf achten, die Datumsanteile beim Erzeugen der `GregorianCalendar`-Objekte korrekt zu setzen – oder Sie erweitern den Algorithmus, so dass er gegebenenfalls selbsttätig den Datumsteil anpasst.

Differenz ohne Berücksichtigung des Tages

Der folgende Algorithmus vergleicht die reinen Uhrzeiten.

- Liegt die Uhrzeit von `time1` zeitlich vor der Uhrzeit von `time2`, wird die Differenz von `time1` zu `time2` berechnet. Beispiel:

  Für `time1` = 07:30 Uhr und `time2` = 22:30 Uhr werden 15 Stunden (bzw. 900 Minuten) berechnet.

- Liegt die Uhrzeit von `time1` zeitlich nach der Uhrzeit von `time2`, wird die Differenz von `time1` zu `time2` am nächsten Tag berechnet. Beispiel:

  Für `time1` = 22:30 Uhr und `time2` = 07:30 Uhr werden 9 Stunden (bzw. 540 Minuten) berechnet.

```
import java.util.Calendar;
import java.util.GregorianCalendar;

GregorianCalendar time1 = new GregorianCalendar(2005, 1, 1, 22, 30, 0);
GregorianCalendar time2 = new GregorianCalendar(2005, 1, 3,  7, 30, 0);

// time1 kopieren und Datumsanteil an time2 angleichen
GregorianCalendar clone1 = (GregorianCalendar) time1.clone();
clone1.set(time2.get(Calendar.YEAR), time2.get(Calendar.MONTH),
           time2.get(Calendar.DAY_OF_MONTH));

// liegt die Uhrzeit von clone1 hinter time2, erhöhe Tag von time2
if (clone1.after(time2))
    time2.add(Calendar.DAY_OF_MONTH, 1);

diff = time2.getTimeInMillis() - clone1.getTimeInMillis();

// Differenz in Millisekunden : diff
// Differenz in Sekunden      : diff/1000
// Differenz in Minuten       : diff/(60*1000)
// Differenz in Stunden       : diff/(60*60*1000)
```

60 Differenz zwischen zwei Uhrzeiten in Stunden, Minuten, Sekunden berechnen

Um die Differenz zwischen zwei Uhrzeiten in eine Kombination aus Stunden, Minuten und Sekunden umzurechnen, berechnen Sie zuerst die Differenz in Sekunden (`diff`). Dann rechnen Sie `diff` Modulo 60 und erhalten den Sekundenanteil. Diesen ziehen Sie von der Gesamtzahl ab (wozu Sie am einfachsten die Ganzzahldivision `diff`/60 durchführen). Analog rechnen Sie den Minutenanteil heraus und behalten die Stunden übrig.

Die statische Methode `getInstance()` der nachfolgend definierten Klasse `TimeDiff` tut genau dies. Sie übernimmt als Argumente die beiden Datumswerte (in Form von `Calendar`-Objekten) sowie ein optionales boolesches Argument, über das sie steuern können, ob die reine Uhrzeitdifferenz ohne Berücksichtigung des Datumsanteils (`true`) oder die Differenz zwischen den vollständigen Datumsangaben (`false`) berechnet wird. Als Ergebnis liefert die Methode ein Objekt ihrer eigenen Klasse zurück, in dessen `public`-Feldern die Werte für Stunden, Minuten und Sekunden gespeichert sind.

```
import java.util.Calendar;

/**
 *  Klasse zur Repräsentation und Berechnung von Zeitabständen
 *  zwischen zwei Uhrzeiten
 *
 */
public class TimeDiff {

    public int hours;
    public int minutes;
    public int seconds;

    // Berechnet die Zeit zwischen zwei Uhrzeiten, gegeben als
    // Calendar-Objekte (berücksichtigt ganzes Datum)
    public static TimeDiff getInstance(Calendar t1, Calendar t2) {
        return getInstance(t1, t2, false);
    }

    // Berechnet die Zeit zwischen zwei Uhrzeiten, gegeben als
    // Calendar-Objekte (wenn onlyClock true, wird nur Differenz zwischen
    // Tageszeiten berechnet)
    public static TimeDiff getInstance(Calendar t1, Calendar t2,
                                       boolean onlyClock) {
        Calendar clone1 = (Calendar) t1.clone();
        long diff;

        // reine Uhrzeit, Datumsanteil eliminieren, vgl. Rezept 59
        if (onlyClock) {
            clone1.set(t2.get(Calendar.YEAR), t2.get(Calendar.MONTH),
                       t2.get(Calendar.DAY_OF_MONTH));
            if (clone1.after(t2))
                t2.add(Calendar.DAY_OF_MONTH, 1);
        }

        diff = Math.abs(t2.getTimeInMillis() - clone1.getTimeInMillis())/1000;

        TimeDiff td = new TimeDiff();

        // Sekunden, Minuten und Stunden berechnen
        td.seconds = (int) (diff%60);  diff /= 60;
        td.minutes = (int) (diff%60);  diff /= 60;
        td.hours   = (int) diff;

        return td;
    }
}
```

Listing 60: Die Klasse TimeDiff

Wenn Sie für den dritten Parameter `false` übergeben oder einfach die überladene Version mit nur zwei Parametern aufrufen, repräsentiert das zurückgelieferte Objekt die Differenz in Stunden, Minuten, Sekunden vom ersten Datum zum zweiten.

Wenn Sie für den dritten Parameter `true` übergeben, repräsentiert das zurückgelieferte Objekt die Stunden, Minuten, Sekunden von der Uhrzeit des ersten `Calendar`-Objekts bis zur Uhrzeit des zweiten `Calendar`-Objekts – so wie die Zeitdifferenz auf dem Zifferblatt einer Uhr abzulesen ist:

- Für `time1` = 07:30 Uhr und `time2` = 22:30 Uhr werden 15 Stunden (bzw. 900 Minuten) berechnet.
- Für `time1` = 22:30 Uhr und `time2` = 07:30 Uhr werden 9 Stunden (bzw. 540 Minuten) berechnet

Das Start-Programm zu diesem Rezept demonstriert die Verwendung:

```
import java.util.Calendar;
import java.util.GregorianCalendar;
import java.text.DateFormat;

public class Start {

    public static void main(String args[]) {
        DateFormat dfDateTime = DateFormat.getDateTimeInstance();
        TimeDiff td;
        System.out.println();

        GregorianCalendar time1 =
                            new GregorianCalendar(2005, 4, 1, 22, 30, 0);
        GregorianCalendar time2 =
                            new GregorianCalendar(2005, 4, 3,  7, 30, 0);
        System.out.println(" Zeit 1 : " + dfDateTime.format(time1.getTime()));
        System.out.println(" Zeit 2 : " + dfDateTime.format(time2.getTime()));


        // Berücksichtigt Datum
        System.out.println("\n Differenz zw. Uhrzeiten (mit Datum)\n");

        td = TimeDiff.getInstance(time1, time2);
        System.out.println(" " + td.hours + " h  "
                           + td.minutes + " min  " + td.seconds + " sec");


        // Reine Uhrzeit
        System.out.println("\n\n Differenz zw. Uhrzeiten (ohne Datum)\n");

        td = TimeDiff.getInstance(time1, time2, true);
        System.out.println(" " + td.hours + " h  "
                           + td.minutes + " min  " + td.seconds + " sec");
    }
}
```

Listing 61: Einsatz der Klasse TimeDiff

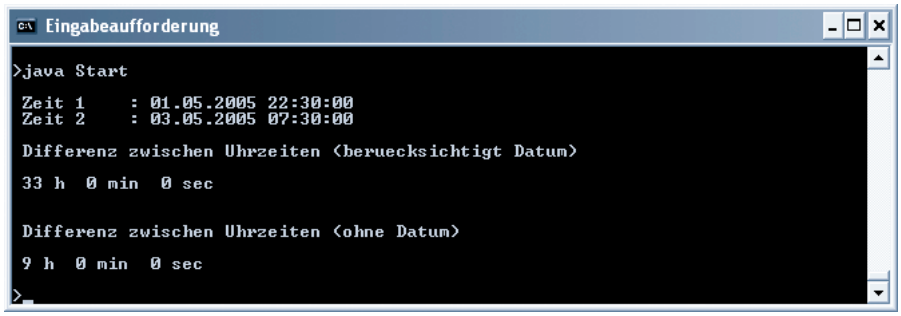


Abbildung 34: Berechnung von Uhrzeitdifferenzen in Stunden, Minuten, Sekunden

61 Präzise Zeitmessungen (Laufzeitmessungen)

Für Zeitmessungen definiert die Klasse `System` die statischen Methoden `currentTimeMillis()` und `nanoTime()`. Beide Methoden werden in gleicher Weise eingesetzt und liefern Zeitwerte in Millisekunden (10^{-3} sec) bzw. Nanosekunden (10^{-9} sec). In der Praxis werden Sie wegen der größeren Genauigkeit in der Regel die ab JDK-Version 1.5 verfügbare Methode `nanoTime()` vorziehen.

Zeitmessungen haben typischerweise folgendes Muster:

```
// 1. Zeitmessung beginnen (Startzeit abfragen)
long start = System.nanoTime();

    // Code, dessen Laufzeit gemessen wird
    Thread.sleep(50000);

// 2. Zeitmessung beenden (Endzeit abfragen)
long end = System.nanoTime();

// 3. Zeitmessung auswerten (Differenz bilden und ausgeben)
long diff = end-start;
System.out.println(" Laufzeit: " + diff);
```

Laufzeitmessungen

Wenn Sie Laufzeitmessungen durchführen, um die Performance eines Algorithmus oder einer Methode zu testen, beachten Sie folgende Punkte:

- Zugriffe auf Konsole, Dateisystem, Internet etc. sollten möglichst vermieden werden.

  Derartige Zugriffe sind oft sehr zeitaufwendig. Wenn Sie einen Algorithmus testen, der Daten aus einer Datei oder Datenbank verarbeitet, messen Sie den Algorithmus unbedingt erst ab dem Zeitpunkt, da die Daten bereits eingelesen sind. Ansonsten kann es passieren, dass das Einlesen der Daten weit mehr Zeit benötigt als deren Verarbeitung und Sie folglich nicht die Effizienz Ihres Algorithmus, sondern die der Einleseoperation messen.

- Benutzeraktionen sollten ebenfalls vermieden werden.

  Sie wollen ja nicht die Reaktionszeit des Benutzers messen, sondern Ihren Code.

Exkurs

currentTimeMillis() und nanoTime()

Die Methode `currentTimeMillis()` gibt es bereits seit dem JDK 1.0. Sie greift, ebenso wie `Date()` oder `Calendar.getInstance()` die aktuelle Systemzeit in Millisekunden seit dem 01.01.1970 00:00:00 Uhr, GMT, ab. (Tatsächlich rufen `Date()` und `Calendar.getInstance()` intern `System.currentTimeMillis()` auf.) Die Genauigkeit von Zeitmessungen mittels `currentTimeMillis()` ist daher von vornherein auf die Größenordnung von Millisekunden beschränkt. Sie verschlechtert sich weiter, wenn der Systemzeitgeber, der die Uhrzeit liefert, in noch längeren Intervallen (etwa alle 10 Millisekunden) aktualisiert wird.

Die Methode `currentTimeMillis()` eignet sich daher nur für Messungen von Operationen, die länger als nur einige Millisekunden andauern (Schreiben in eine Datei, Zugriff auf Datenbanken oder Internet, Messung der Zeit, die ein Benutzer für die Bearbeitung eines Dialogfelds oder Ähnliches benötigt).

Die Methode `nanoTime()` gibt es erst seit dem JDK 1.5. Sie fragt die Zeit von dem genauestens verfügbaren Systemzeitgeber ab. (Die meisten Rechner besitzen mittlerweile Systemzeitgeber, die im Bereich von Nanosekunden aktualisiert werden.) Die von diesen Systemzeitgebern zurückgelieferte Anzahl Nanosekunden muss sich allerdings nicht auf eine feste Zeit beziehen und kann/sollte daher nicht als Zeit/Datum interpretiert werden. (Versuchen Sie also nicht, den Rückgabewert von `nanoTime()` in Millisekunden umzurechnen und zum Setzen eines `Date`- oder `Calendar`-Objekts zu verwenden.)

Die Methode `nanoTime()` ist die Methode der Wahl für Performance-Messungen.

- Führen Sie wiederholte Messungen durch.

  Verlassen Sie sich nie auf eine Messung. Wiederholen Sie die Messungen, beispielsweise in einer Schleife, und bilden Sie den Mittelwert.

- Verwenden Sie stets gleiche Ausgangsdaten.

  Wenn Sie verschiedene Algorithmen/Methoden miteinander vergleichen, achten Sie darauf, dass die Tests unter denselben Bedingungen und mit denselben Ausgangsdaten durchgeführt werden.

Achtung

Vorsicht Jitter! Viele Java-Interpreter fallen unter die Kategorie der Just-In-Time-Compiler, insofern als sie Codeblöcke wie z.B. Methoden bei der ersten Ausführung von Bytecode in Maschinencode umwandeln, speichern und bei der nächsten Ausführung dann den bereits vorliegenden Maschinencode ausführen. In diesem Fall sollten Sie eine zu beurteilende Methode unbedingt mehrfach ausführen und die erste Laufzeitmessung verwerfen.

Nanosekunden in Stunden, Minuten, Sekunden, Millisekunden und Nanosekunden umrechnen

Laufzeitunterschiede, die in Nanosekunden ausgegeben werden, können vom Menschen meist nur schwer miteinander verglichen und in ihrer tatsächlichen Größenordnung erfasst werden. Es bietet sich daher an, die in Nanosekunden berechnete Differenz vor der Ausgabe in eine Kombination höherer Einheiten umzurechnen.

```
/**
 *  Klasse zum Umrechnen von Nanosekunden in Stunden, Minuten...
 *
 */
public class TimeDiff {

    public int hours;
    public int minutes;
    public int seconds;
    public int millis;
    public int nanos;

    public static TimeDiff getInstance(long time) {
        TimeDiff td = new TimeDiff();

        td.nanos   = (int) (time%1000000);  time /= 1000000;
        td.millis  = (int) (time%1000);  time /= 1000;
        td.seconds = (int) (time%60);  time/= 60;
        td.minutes = (int) (time%60);  time/= 60;
        td.hours   = (int) time;

        return td;
    }
}
```

Listing 62: Die Klasse TimeDiff zerlegt eine Nanosekunden-Angabe in höhere Einheiten.

Aufruf:

```
// 3. Zeitmessung auswerten (Differenz bilden und ausgeben)
long diff = end-start;
td = TimeDiff.getInstance(diff);
System.out.println(" Laufzeit: " + td.hours + " h  "
                   + td.minutes + " min  " + td.seconds + " sec  "
                   + td.millis + " milli  " + td.nanos + " nano");
```

62 Uhrzeit einblenden

In den bisherigen Rezepten ging es mehr oder weniger immer darum, die Zeit einmalig abzufragen und irgendwie weiterzuverarbeiten. Wie aber sieht es aus, wenn die Uhrzeit als digitale Zeitanzeige in die Oberfläche einer GUI-Anwendung oder eines Applets eingeblendet werden soll?

Zur Erzeugung einer Uhr müssen Sie die Zeit kontinuierlich abfragen und ausgeben. In diesem Rezept erfolgen das Abfragen und das Anzeigen der Zeit weitgehend getrennt.

- Für das Abfragen ist eine Klasse `ClockThread` verantwortlich, die, wie der Name schon verrät, von Thread abgeleitet ist und einen eigenständigen Thread repräsentiert.
- Die Anzeige der Uhr kann in einer beliebigen Swing-Komponente (zurückgehend auf die Basisklasse `JComponent`) erfolgen.

Um die Verbindung zwischen `ClockThread` und Swing-Komponente herzustellen, übernimmt der `ClockThread`-Konstruktor eine Referenz auf die Komponente. Als Dank fordert er die Komponente nach jeder Aktualisierung der Uhrzeit auf, sich neu zu zeichnen.

```
import java.util.Date;
import java.text.DateFormat;
import javax.swing.JComponent;

/**
 * Thread-Klasse, die aktuelle Uhrzeit in Komponenten einblendet
 *
 */
public class ClockThread extends Thread {

    private static String time;
    private DateFormat df = DateFormat.getTimeInstance();
    private JComponent c;

    public ClockThread(JComponent c) {
        this.c = c;
        this.start();
    }

    public void run() {
       while(isInterrupted() == false) {

          // Uhrzeit aktualisieren
          ClockThread.time = df.format(new Date());

          // Komponente zum Neuzeichnen auffordern
          c.repaint();

          // eine Sekunde schlafen
          try {
             sleep(1000);
          }
          catch(InterruptedException e)  {
             return;
          }
       }
    }

    public static String getTime() {
       return time;
    }
}
```

Listing 63: Die Klasse ClockThread

Der Konstruktor von `ClockThread` speichert die Referenz auf die Anzeige-Komponente und startet den Thread, woraufhin intern dessen `run()`-Methode gestartet wird (mehr zu Threads in der Kategorie »Threads«). Die `run()`-Methode enthält eine einzige große `while`-Schleife, die so lange durchlaufen wird, wie der Thread ausgeführt wird. In der Schleife wird die aktuelle Zeit abgefragt, formatiert und im statischen Feld `time` gespeichert, von wo sie die Anzeige-Komponente mit Hilfe der `public getTime()`-Methode auslesen kann.

Das Start-Programm zu diesem Rezept demonstriert, wie die Uhrzeit mit Hilfe von `ClockThread` in einem `JPanel`, hier die `ContentPane` des Fensters, angezeigt werden kann.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Start extends JFrame {

   class ClockPanel extends JPanel {
       public void paintComponent(Graphics g) {
           super.paintComponent(g);

           // Uhrzeit einblenden
           g.setFont(new Font("Arial", Font.PLAIN, 18));
           g.setColor(Color.blue);
           g.drawString(ClockThread.getTime(), 15, 30);
       }
   }

   private ClockThread ct;
   private ClockPanel display;

   public Start() {
       setTitle("Fenster mit Uhrzeit");
       display = new ClockPanel();
       getContentPane().add(display, BorderLayout.CENTER);
       setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

       // Thread für Uhrzeit erzeugen und starten
       ct = new ClockThread(display);
   }

   public static void main(String args[]) {
      // Fenster erzeugen und anzeigen
      Start  mw = new Start();
      mw.setSize(500,350);
      mw.setLocation(200,300);
      mw.setVisible(true);
   }
}
```

Listing 64: GUI-Programm mit Uhreinblendung

Dem Fenster fällt die Aufgabe zu, den Uhrzeit-Thread in Gang zu setzen und mit der Anzeige-Komponente zu verbinden. Beides geschieht im Konstruktor des Fensters bei der Erzeugung des `ClockThread`-Objekts.

Für die Anzeige-Komponente muss eine eigene Klasse (`ClockPanel`) abgeleitet werden. Nur so ist es möglich, die `paintComponent()`-Methode zu überschreiben und den Code zum Einblenden der Uhrzeit aufzunehmen.

Abbildung 35: GUI-Programm mit eingeblendeter Uhrzeit in JPanel