














 openjdk-mirror /  
**jdk7u-jdk**




 **Code**  Issues 2  Pull requests 2  Actions  Projects  Wiki  Security 

 master **jdk7u-jdk / src / share / classes / java / util**

 Go to file t 

**/ Date.java** 

**ohair** 6962318: Update copyright year 14 years ago 

1331 lines (1272 loc) · 55.2 KB

```
1  /*
2   * Copyright (c) 1994, 2010, Oracle and/or its affiliates. All rights reserved.
3   * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
4   *
5   * This code is free software; you can redistribute it and/or modify it
6   * under the terms of the GNU General Public License version 2 only, as
7   * published by the Free Software Foundation. Oracle designates this
8   * particular file as subject to the "Classpath" exception as provided
9   * by Oracle in the LICENSE file that accompanied this code.
10  *
11  * This code is distributed in the hope that it will be useful, but WITHOUT
12  * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
13  * FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License
14  * version 2 for more details (a copy is included in the LICENSE file that
15  * accompanied this code).
16  *
17  * You should have received a copy of the GNU General Public License version
18  * 2 along with this work; if not, write to the Free Software Foundation,
19  * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
20  *
21  * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
22  * or visit www.oracle.com if you need additional information or have any
23  * questions.
24  */
25
26  package java.util;
27
28  import java.text.DateFormat;
29  import java.io.IOException;
30  import java.io.ObjectOutputStream;
31  import java.io.ObjectInputStream;
32  import java.lang.ref.SoftReference;
33  import sun.util.calendar.BaseCalendar;
34  import sun.util.calendar.CalendarDate;
35  import sun.util.calendar.CalendarSystem;
36  import sun.util.calendar.CalendarUtils;
37  import sun.util.calendar.Era;
```

```
38 import sun.util.calendar.Gregorian;
39 import sun.util.calendar.ZoneInfo;
40
41 /**
42  * The class Date represents a specific instant
43  * in time, with millisecond precision.
44  * <p>
45  * Prior to JDK&nbsp;1.1, the class Date had two additional
46  * functions. It allowed the interpretation of dates as year, month, day, hour,
47  * minute, and second values. It also allowed the formatting and parsing
48  * of date strings. Unfortunately, the API for these functions was not
49  * amenable to internationalization. As of JDK&nbsp;1.1, the
50  * Calendar class should be used to convert between dates and time
51  * fields and the DateFormat class should be used to format and
52  * parse date strings.
53  * The corresponding methods in Date are deprecated.
54  * <p>
55  * Although the Date class is intended to reflect
56  * coordinated universal time (UTC), it may not do so exactly,
57  * depending on the host environment of the Java Virtual Machine.
58  * Nearly all modern operating systems assume that 1&nbsp;day&nbsp;=
59  * 24&nbsp;&times;&nbsp;60&nbsp;&times;&nbsp;60&nbsp;= 86400 seconds
60  * in all cases. In UTC, however, about once every year or two there
61  * is an extra second, called a "leap second." The leap
62  * second is always added as the last second of the day, and always
63  * on December 31 or June 30. For example, the last minute of the
64  * year 1995 was 61 seconds long, thanks to an added leap second.
65  * Most computer clocks are not accurate enough to be able to reflect
66  * the leap-second distinction.
67  * <p>
68  * Some computer standards are defined in terms of Greenwich mean
69  * time (GMT), which is equivalent to universal time (UT). GMT is
70  * the "civil" name for the standard; UT is the
71  * "scientific" name for the same standard. The
72  * distinction between UTC and UT is that UTC is based on an atomic
73  * clock and UT is based on astronomical observations, which for all
74  * practical purposes is an invisibly fine hair to split. Because the
75  * earth's rotation is not uniform (it slows down and speeds up
76  * in complicated ways), UT does not always flow uniformly. Leap
77  * seconds are introduced as needed into UTC so as to keep UTC within
78  * 0.9 seconds of UT1, which is a version of UT with certain
79  * corrections applied. There are other time and date systems as
80  * well; for example, the time scale used by the satellite-based
81  * global positioning system (GPS) is synchronized to UTC but is
82  * <i>not</i> adjusted for leap seconds. An interesting source of
83  * further information is the U.S. Naval Observatory, particularly
84  * the Directorate of Time at:
85  * <blockquote><pre>
86  *     <a href=http://tycho.usno.navy.mil>http://tycho.usno.navy.mil</a>
87  * </pre></blockquote>
88  * <p>
89  * and their definitions of "Systems of Time" at:
90  * <blockquote><pre>
91  *     <a href=http://tycho.usno.navy.mil/systime.html>http://tycho.usno.navy.mil/systime.html<
92  * </pre></blockquote>
```

```

93      * <p>
94      * In all methods of class <code>Date</code> that accept or return
95      * year, month, date, hours, minutes, and seconds values, the
96      * following representations are used:
97      * <ul>
98      * <li>A year <i>y</i> is represented by the integer
99      *   <i>y</i>&nbsp;<code>-&nbsp;1900</code>.
100     * <li>A month is represented by an integer from 0 to 11; 0 is January,
101     *   1 is February, and so forth; thus 11 is December.
102     * <li>A date (day of month) is represented by an integer from 1 to 31
103     *   in the usual manner.
104     * <li>An hour is represented by an integer from 0 to 23. Thus, the hour
105     *   from midnight to 1 a.m. is hour 0, and the hour from noon to 1
106     *   p.m. is hour 12.
107     * <li>A minute is represented by an integer from 0 to 59 in the usual manner.
108     * <li>A second is represented by an integer from 0 to 61; the values 60 and
109     *   61 occur only for leap seconds and even then only in Java
110     *   implementations that actually track leap seconds correctly. Because
111     *   of the manner in which leap seconds are currently introduced, it is
112     *   extremely unlikely that two leap seconds will occur in the same
113     *   minute, but this specification follows the date and time conventions
114     *   for ISO C.
115     * </ul>
116     * <p>
117     * In all cases, arguments given to methods for these purposes need
118     * not fall within the indicated ranges; for example, a date may be
119     * specified as January 32 and is interpreted as meaning February 1.
120     *
121     * @author   James Gosling
122     * @author   Arthur van Hoff
123     * @author   Alan Liu
124     * @see      java.text.DateFormat
125     * @see      java.util.Calendar
126     * @see      java.util.TimeZone
127     * @since    JDK1.0
128     */
129     ✓ public class Date
130         implements java.io.Serializable, Cloneable, Comparable<Date>
131     {
132         private static final BaseCalendar gcal =
133             CalendarSystem.getGregorianCalendar();
134         private static BaseCalendar jcal;
135
136         private transient long fastTime;
137
138         /*
139          * If cdate is null, then fastTime indicates the time in millis.
140          * If cdate.isNormalized() is true, then fastTime and cdate are in
141          * synch. Otherwise, fastTime is ignored, and cdate indicates the
142          * time.
143          */
144         private transient BaseCalendar.Date cdate;
145
146         // Initialized just before the value is used. See parse().
147         private static int defaultCenturyStart;

```

```
147     private static int defaultCenturyStart;
148
149     /* use serialVersionUID from modified java.util.Date for
150      * interoperability with JDK1.1. The Date was modified to write
151      * and read only the UTC time.
152      */
153     private static final long serialVersionUID = 7523967970034938905L;
154
155     /**
156      * Allocates a <code>Date</code> object and initializes it so that
157      * it represents the time at which it was allocated, measured to the
158      * nearest millisecond.
159      *
160      * @see      java.lang.System#currentTimeMillis()
161      */
162     public Date() {
163         this(System.currentTimeMillis());
164     }
165
166     /**
```



master ▾

jdk7u-jdk / src / share / classes / java / util / Date.java

↑ Top

Code

Blame

Raw



```
129     public class Date
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173     * @see      java.lang.System#currentTimeMillis()
174     */
175     public Date(long date) {
176         fastTime = date;
177     }
178
179     /**
180      * Allocates a <code>Date</code> object and initializes it so that
181      * it represents midnight, local time, at the beginning of the day
182      * specified by the <code>year</code>, <code>month</code>, and
183      * <code>date</code> arguments.
184      *
185      * @param   year    the year minus 1900.
186      * @param   month    the month between 0-11.
187      * @param   date     the day of the month between 1-31.
188      * @see      java.util.Calendar
189      * @deprecated As of JDK version 1.1,
190      * replaced by <code>Calendar.set(year + 1900, month, date)</code>
191      * or <code>GregorianCalendar(year + 1900, month, date)</code>.
192      */
193     @Deprecated
194     public Date(int year, int month, int date) {
195         this(year, month, date, 0, 0, 0);
196     }
197
198     /**
199      * Allocates a <code>Date</code> object and initializes it so that
200      * it represents the instant at the start of the minute specified by
201      * the <code>year</code>, <code>month</code>, <code>date</code>,
```

```
202 * <code>hrs</code>, and <code>min</code> arguments, in the local
203 * time zone.
204 *
205 * @param year the year minus 1900.
206 * @param month the month between 0-11.
207 * @param date the day of the month between 1-31.
208 * @param hrs the hours between 0-23.
209 * @param min the minutes between 0-59.
210 * @see java.util.Calendar
211 * @deprecated As of JDK version 1.1,
212 * replaced by <code>Calendar.set(year + 1900, month, date,
213 * hrs, min)</code> or <code>GregorianCalendar(year + 1900,
214 * month, date, hrs, min)</code>.
215 */
216 @Deprecated
217 public Date(int year, int month, int date, int hrs, int min) {
218     this(year, month, date, hrs, min, 0);
219 }
220
221 /**
222 * Allocates a <code>Date</code> object and initializes it so that
223 * it represents the instant at the start of the second specified
224 * by the <code>year</code>, <code>month</code>, <code>date</code>,
225 * <code>hrs</code>, <code>min</code>, and <code>sec</code> arguments,
226 * in the local time zone.
227 *
228 * @param year the year minus 1900.
229 * @param month the month between 0-11.
230 * @param date the day of the month between 1-31.
231 * @param hrs the hours between 0-23.
232 * @param min the minutes between 0-59.
233 * @param sec the seconds between 0-59.
234 * @see java.util.Calendar
235 * @deprecated As of JDK version 1.1,
236 * replaced by <code>Calendar.set(year + 1900, month, date,
237 * hrs, min, sec)</code> or <code>GregorianCalendar(year + 1900,
238 * month, date, hrs, min, sec)</code>.
239 */
240 @Deprecated
241 public Date(int year, int month, int date, int hrs, int min, int sec) {
242     int y = year + 1900;
243     // month is 0-based. So we have to normalize month to support Long.MAX_VALUE.
244     if (month >= 12) {
245         y += month / 12;
246         month %= 12;
247     } else if (month < 0) {
248         y += CalendarUtils.floorDivide(month, 12);
249         month = CalendarUtils.mod(month, 12);
250     }
251     BaseCalendar cal = getCalendarSystem(y);
252     cdate = (BaseCalendar.Date) cal.newCalendarDate(TimeZone.getDefaultRef());
253     cdate.setNormalizedDate(y, month + 1, date).setTimeOfDay(hrs, min, sec, 0);
254     getTimeImpl();
255     cdate = null;
256 }
```

```
257
258 /**
259  * Allocates a Date object and initializes it so that
260  * it represents the date and time indicated by the string
261  * s, which is interpreted as if by the
262  * {@link Date#parse} method.
263  *
264  * @param s a string representation of the date.
265  * @see java.text.DateFormat
266  * @see java.util.Date#parse(java.lang.String)
267  * @deprecated As of JDK version 1.1,
268  * replaced by DateFormat.parse(String s).
269  */
270 @Deprecated
271 public Date(String s) {
272     this(parse(s));
273 }
274
275 /**
276  * Return a copy of this object.
277  */
278 public Object clone() {
279     Date d = null;
280     try {
281         d = (Date)super.clone();
282         if (cdate != null) {
283             d.cdate = (BaseCalendar.Date) cdate.clone();
284         }
285     } catch (CloneNotSupportedException e) {} // Won't happen
286     return d;
287 }
288
289 /**
290  * Determines the date and time based on the arguments. The
291  * arguments are interpreted as a year, month, day of the month,
292  * hour of the day, minute within the hour, and second within the
293  * minute, exactly as for the Date constructor with six
294  * arguments, except that the arguments are interpreted relative
295  * to UTC rather than to the local time zone. The time indicated is
296  * returned represented as the distance, measured in milliseconds,
297  * of that time from the epoch (00:00:00 GMT on January 1, 1970).
298  *
299  * @param year the year minus 1900.
300  * @param month the month between 0-11.
301  * @param date the day of the month between 1-31.
302  * @param hrs the hours between 0-23.
303  * @param min the minutes between 0-59.
304  * @param sec the seconds between 0-59.
305  * @return the number of milliseconds since January 1, 1970, 00:00:00 GMT for
306  * the date and time specified by the arguments.
307  * @see java.util.Calendar
308  * @deprecated As of JDK version 1.1,
309  * replaced by Calendar.set(year + 1900, month, date,
310  * hrs, min, sec) or GregorianCalendar(year + 1900,
```

```

311 * month, date, hrs, min, sec)</code>, using a UTC
312 * <code>TimeZone</code>, followed by <code>Calendar.getTime().getTime()</code>.
313 */
314 @Deprecated
315 public static long UTC(int year, int month, int date,
316                        int hrs, int min, int sec) {
317     int y = year + 1900;
318     // month is 0-based. So we have to normalize month to support Long.MAX_VALUE.
319     if (month >= 12) {
320         y += month / 12;
321         month %= 12;
322     } else if (month < 0) {
323         y += CalendarUtils.floorDivide(month, 12);
324         month = CalendarUtils.mod(month, 12);
325     }
326     int m = month + 1;
327     BaseCalendar cal = getCalendarSystem(y);
328     BaseCalendar.Date udate = (BaseCalendar.Date) cal.newCalendarDate(null);
329     udate.setNormalizedDate(y, m, date).setTimeOfDay(hrs, min, sec, 0);
330
331     // Use a Date instance to perform normalization. Its fastTime
332     // is the UTC value after the normalization.
333     Date d = new Date(0);
334     d.normalize(udate);
335     return d.fastTime;
336 }
337
338 /**
339  * Attempts to interpret the string <tt>s</tt> as a representation
340  * of a date and time. If the attempt is successful, the time
341  * indicated is returned represented as the distance, measured in
342  * milliseconds, of that time from the epoch (00:00:00 GMT on
343  * January 1, 1970). If the attempt fails, an
344  * <tt>IllegalArgumentException</tt> is thrown.
345  * <p>
346  * It accepts many syntaxes; in particular, it recognizes the IETF
347  * standard date syntax: "Sat, 12 Aug 1995 13:30:00 GMT". It also
348  * understands the continental U.S. time-zone abbreviations, but for
349  * general use, a time-zone offset should be used: "Sat, 12 Aug 1995
350  * 13:30:00 GMT+0430" (4 hours, 30 minutes west of the Greenwich
351  * meridian). If no time zone is specified, the local time zone is
352  * assumed. GMT and UTC are considered equivalent.
353  * <p>
354  * The string <tt>s</tt> is processed from left to right, looking for
355  * data of interest. Any material in <tt>s</tt> that is within the
356  * ASCII parenthesis characters <tt>(</tt> and <tt>)</tt> is ignored.
357  * Parentheses may be nested. Otherwise, the only characters permitted
358  * within <tt>s</tt> are these ASCII characters:
359  * <blockquote><pre>
360  * abcdefghijklmnopqrstuvwxyz
361  * ABCDEFGHIJKLMNOPQRSTUVWXYZ
362  * 0123456789,+-.:/</pre></blockquote>
363  * and whitespace characters.<p>
364  * A consecutive sequence of decimal digits is treated as a decimal
365  * number:<ul>

```

```

366 * <li>If a number is preceded by <tt>+</tt> or <tt>-</tt> and a year
367 *     has already been recognized, then the number is a time-zone
368 *     offset. If the number is less than 24, it is an offset measured
369 *     in hours. Otherwise, it is regarded as an offset in minutes,
370 *     expressed in 24-hour time format without punctuation. A
371 *     preceding <tt>-</tt> means a westward offset. Time zone offsets
372 *     are always relative to UTC (Greenwich). Thus, for example,
373 *     <tt>-5</tt> occurring in the string would mean "five hours west
374 *     of Greenwich" and <tt>+0430</tt> would mean "four hours and
375 *     thirty minutes east of Greenwich." It is permitted for the
376 *     string to specify <tt>GMT</tt>, <tt>UT</tt>, or <tt>UTC</tt>
377 *     redundantly-for example, <tt>GMT-5</tt> or <tt>utc+0430</tt>.
378 * <li>The number is regarded as a year number if one of the
379 *     following conditions is true:
380 * <ul>
381 *     <li>The number is equal to or greater than 70 and followed by a
382 *         space, comma, slash, or end of string
383 *     <li>The number is less than 70, and both a month and a day of
384 *         the month have already been recognized</li>
385 * </ul>
386 *     If the recognized year number is less than 100, it is
387 *     interpreted as an abbreviated year relative to a century of
388 *     which dates are within 80 years before and 19 years after
389 *     the time when the Date class is initialized.
390 *     After adjusting the year number, 1900 is subtracted from
391 *     it. For example, if the current year is 1999 then years in
392 *     the range 19 to 99 are assumed to mean 1919 to 1999, while
393 *     years from 0 to 18 are assumed to mean 2000 to 2018. Note
394 *     that this is slightly different from the interpretation of
395 *     years less than 100 that is used in {@link java.text.SimpleDateFormat}.
396 * <li>If the number is followed by a colon, it is regarded as an hour,
397 *     unless an hour has already been recognized, in which case it is
398 *     regarded as a minute.
399 * <li>If the number is followed by a slash, it is regarded as a month
400 *     (it is decreased by 1 to produce a number in the range <tt>0</tt>
401 *     to <tt>11</tt>), unless a month has already been recognized, in
402 *     which case it is regarded as a day of the month.
403 * <li>If the number is followed by whitespace, a comma, a hyphen, or
404 *     end of string, then if an hour has been recognized but not a
405 *     minute, it is regarded as a minute; otherwise, if a minute has
406 *     been recognized but not a second, it is regarded as a second;
407 *     otherwise, it is regarded as a day of the month. </ul><p>
408 * A consecutive sequence of letters is regarded as a word and treated
409 * as follows:<ul>
410 * <li>A word that matches <tt>AM</tt>, ignoring case, is ignored (but
411 *     the parse fails if an hour has not been recognized or is less
412 *     than <tt>1</tt> or greater than <tt>12</tt>).
413 * <li>A word that matches <tt>PM</tt>, ignoring case, adds <tt>12</tt>
414 *     to the hour (but the parse fails if an hour has not been
415 *     recognized or is less than <tt>1</tt> or greater than <tt>12</tt>).
416 * <li>Any word that matches any prefix of <tt>SUNDAY, MONDAY, TUESDAY,
417 *     WEDNESDAY, THURSDAY, FRIDAY</tt>, or <tt>SATURDAY</tt>, ignoring
418 *     case, is ignored. For example, <tt>sat, Friday, TUE</tt>, and
419 *     <tt>Thurs</tt> are ignored.
420 * <li>Otherwise, any word that matches any prefix of <tt>JANUARY

```



```

420 * <li>Otherwise, any word that matches any prefix of <tt>JANUARY,
421 *     FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER,
422 *     OCTOBER, NOVEMBER</tt>, or <tt>DECEMBER</tt>, ignoring case, and
423 *     considering them in the order given here, is recognized as
424 *     specifying a month and is converted to a number (<tt>0</tt> to
425 *     <tt>11</tt>). For example, <tt>aug, Sept, april</tt>, and
426 *     <tt>NOV</tt> are recognized as months. So is <tt>Ma</tt>, which
427 *     is recognized as <tt>MARCH</tt>, not <tt>MAY</tt>.
428 * <li>Any word that matches <tt>GMT, UT</tt>, or <tt>UTC</tt>, ignoring
429 *     case, is treated as referring to UTC.
430 * <li>Any word that matches <tt>EST, CST, MST</tt>, or <tt>PST</tt>,
431 *     ignoring case, is recognized as referring to the time zone in
432 *     North America that is five, six, seven, or eight hours west of
433 *     Greenwich, respectively. Any word that matches <tt>EDT, CDT,
434 *     MDT</tt>, or <tt>PDT</tt>, ignoring case, is recognized as
435 *     referring to the same time zone, respectively, during daylight
436 *     saving time.</ul><p>
437 * Once the entire string s has been scanned, it is converted to a time
438 * result in one of two ways. If a time zone or time-zone offset has been
439 * recognized, then the year, month, day of month, hour, minute, and
440 * second are interpreted in UTC and then the time-zone offset is
441 * applied. Otherwise, the year, month, day of month, hour, minute, and
442 * second are interpreted in the local time zone.
443 *
444 * @param   s      a string to be parsed as a date.
445 * @return  the number of milliseconds since January 1, 1970, 00:00:00 GMT
446 *          represented by the string argument.
447 * @see     java.text.DateFormat
448 * @deprecated As of JDK version 1.1,
449 * replaced by <code>DateFormat.parse(String s)</code>.
450 */
451 @Deprecated
452 public static long parse(String s) {
453     int year = Integer.MIN_VALUE;
454     int mon = -1;
455     int mday = -1;
456     int hour = -1;
457     int min = -1;
458     int sec = -1;
459     int millis = -1;
460     int c = -1;
461     int i = 0;
462     int n = -1;
463     int wst = -1;
464     int tzoffset = -1;
465     int prevc = 0;
466     syntax:
467     {
468         if (s == null)
469             break syntax;
470         int limit = s.length();
471         while (i < limit) {
472             c = s.charAt(i);
473             i++;
474             if (c <= ' ' || c == ',')

```

```

475         continue;
476     if (c == '(') { // skip comments
477         int depth = 1;
478         while (i < limit) {
479             c = s.charAt(i);
480             i++;
481             if (c == '(') depth++;
482             else if (c == ')')
483                 if (--depth <= 0)
484                     break;
485         }
486         continue;
487     }
488     if ('0' <= c && c <= '9') {
489         n = c - '0';
490         while (i < limit && '0' <= (c = s.charAt(i)) && c <= '9') {
491             n = n * 10 + c - '0';
492             i++;
493         }
494         if (prevc == '+' || prevc == '-' && year != Integer.MIN_VALUE) {
495             // timezone offset
496             if (n < 24)
497                 n = n * 60; // EG. "GMT-3"
498             else
499                 n = n % 100 + n / 100 * 60; // eg "GMT-0430"
500             if (prevc == '+') // plus means east of GMT
501                 n = -n;
502             if (tzoffset != 0 && tzoffset != -1)
503                 break syntax;
504             tzoffset = n;
505         } else if (n >= 70)
506             if (year != Integer.MIN_VALUE)
507                 break syntax;
508             else if (c <= ' ' || c == ',' || c == '/' || i >= limit)
509                 // year = n < 1900 ? n : n - 1900;
510                 year = n;
511             else
512                 break syntax;
513         else if (c == ':')
514             if (hour < 0)
515                 hour = (byte) n;
516             else if (min < 0)
517                 min = (byte) n;
518             else
519                 break syntax;
520         else if (c == '/')
521             if (mon < 0)
522                 mon = (byte) (n - 1);
523             else if (mday < 0)
524                 mday = (byte) n;
525             else
526                 break syntax;
527         else if (i < limit && c != ',' && c > ' ' && c != '-')
528             break syntax;
529         else if (hour >= 0 && min < 0)

```

```

530         min = (byte) n;
531     else if (min >= 0 && sec < 0)
532         sec = (byte) n;
533     else if (mday < 0)
534         mday = (byte) n;
535     // Handle two-digit years < 70 (70-99 handled above).
536     else if (year == Integer.MIN_VALUE && mon >= 0 && mday >= 0)
537         year = n;
538     else
539         break syntax;
540     prevc = 0;
541 } else if (c == '/' || c == ':' || c == '+' || c == '-')
542     prevc = c;
543 else {
544     int st = i - 1;
545     while (i < limit) {
546         c = s.charAt(i);
547         if (!('A' <= c && c <= 'Z' || 'a' <= c && c <= 'z'))
548             break;
549         i++;
550     }
551     if (i <= st + 1)
552         break syntax;
553     int k;
554     for (k = wtb.length; --k >= 0;)
555         if (wtb[k].regionMatches(true, 0, s, st, i - st)) {
556             int action = ttb[k];
557             if (action != 0) {
558                 if (action == 1) { // pm
559                     if (hour > 12 || hour < 1)
560                         break syntax;
561                     else if (hour < 12)
562                         hour += 12;
563                 } else if (action == 14) { // am
564                     if (hour > 12 || hour < 1)
565                         break syntax;
566                     else if (hour == 12)
567                         hour = 0;
568                 } else if (action <= 13) { // month!
569                     if (mon < 0)
570                         mon = (byte) (action - 2);
571                     else
572                         break syntax;
573                 } else {
574                     tzoffset = action - 10000;
575                 }
576             }
577             break;
578         }
579     if (k < 0)
580         break syntax;
581     prevc = 0;
582 }
583 }

```

```

584         if (year == Integer.MIN_VALUE || mon < 0 || mday < 0)
585             break syntax;
586         // Parse 2-digit years within the correct default century.
587         if (year < 100) {
588             synchronized (Date.class) {
589                 if (defaultCenturyStart == 0) {
590                     defaultCenturyStart = gcal.getCalendarDate().getYear() - 80;
591                 }
592             }
593             year += (defaultCenturyStart / 100) * 100;
594             if (year < defaultCenturyStart) year += 100;
595         }
596         if (sec < 0)
597             sec = 0;
598         if (min < 0)
599             min = 0;
600         if (hour < 0)
601             hour = 0;
602         BaseCalendar cal = getCalendarSystem(year);
603         if (tzoffset == -1) { // no time zone specified, have to use local
604             BaseCalendar.Date ldate = (BaseCalendar.Date) cal.newCalendarDate(TimeZone.getDT
605                 ldate.setDate(year, mon + 1, mday);
606                 ldate.setTimeOfDay(hour, min, sec, 0);
607                 return cal.getTime(ldate);
608         }
609         BaseCalendar.Date udate = (BaseCalendar.Date) cal.newCalendarDate(null); // no time
610         udate.setDate(year, mon + 1, mday);
611         udate.setTimeOfDay(hour, min, sec, 0);
612         return cal.getTime(udate) + tzoffset * (60 * 1000);
613     }
614     // syntax error
615     throw new IllegalArgumentException();
616 }
617 private final static String wtb[] = {
618     "am", "pm",
619     "monday", "tuesday", "wednesday", "thursday", "friday",
620     "saturday", "sunday",
621     "january", "february", "march", "april", "may", "june",
622     "july", "august", "september", "october", "november", "december",
623     "gmt", "ut", "utc", "est", "edt", "cst", "cdt",
624     "mst", "mdt", "pst", "pdt"
625 };
626 private final static int ttb[] = {
627     14, 1, 0, 0, 0, 0, 0, 0, 0, 0,
628     2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
629     10000 + 0, 10000 + 0, 10000 + 0, // GMT/UT/UTC
630     10000 + 5 * 60, 10000 + 4 * 60, // EST/EDT
631     10000 + 6 * 60, 10000 + 5 * 60, // CST/CDT
632     10000 + 7 * 60, 10000 + 6 * 60, // MST/MDT
633     10000 + 8 * 60, 10000 + 7 * 60 // PST/PDT
634 };
635
636 /**
637  * Returns a value that is the result of subtracting 1900 from the
638  * year that contains or begins with the instant in time represented

```

```
639      * by this Date object, as interpreted in the local
640      * time zone.
641      *
642      * @return the year represented by this date, minus 1900.
643      * @see java.util.Calendar
644      * @deprecated As of JDK version 1.1,
645      * replaced by Calendar.get(Calendar.YEAR) - 1900.
646      */
647      @Deprecated
648      public int getYear() {
649          return normalize().getYear() - 1900;
650      }
651
652      /**
653       * Sets the year of this Date object to be the specified
654       * value plus 1900. This Date object is modified so
655       * that it represents a point in time within the specified year,
656       * with the month, date, hour, minute, and second the same as
657       * before, as interpreted in the local time zone. (Of course, if
658       * the date was February 29, for example, and the year is set to a
659       * non-leap year, then the new date will be treated as if it were
660       * on March 1.)
661       *
662       * @param year the year value.
663       * @see java.util.Calendar
664       * @deprecated As of JDK version 1.1,
665       * replaced by Calendar.set(Calendar.YEAR, year + 1900).
666       */
667      @Deprecated
668      public void setYear(int year) {
669          getCalendarDate().setNormalizedYear(year + 1900);
670      }
671
672      /**
673       * Returns a number representing the month that contains or begins
674       * with the instant in time represented by this Date object.
675       * The value returned is between 0 and 11,
676       * with the value 0 representing January.
677       *
678       * @return the month represented by this date.
679       * @see java.util.Calendar
680       * @deprecated As of JDK version 1.1,
681       * replaced by Calendar.get(Calendar.MONTH).
682       */
683      @Deprecated
684      public int getMonth() {
685          return normalize().getMonth() - 1; // adjust 1-based to 0-based
686      }
687
688      /**
689       * Sets the month of this date to the specified value. This
690       * Date object is modified so that it represents a point
691       * in time within the specified month, with the year, date, hour,
692       * minute, and second the same as before, as interpreted in the
693       * local time zone. If the date was October 31, for example, and
```

```
693     * local time zone. If the date was October 31, for example, and
694     * the month is set to June, then the new date will be treated as
695     * if it were on July 1, because June has only 30 days.
696     *
697     * @param month the month value between 0-11.
698     * @see java.util.Calendar
699     * @deprecated As of JDK version 1.1,
700     * replaced by Calendar.set(Calendar.MONTH, int month).
701     */
702     @Deprecated
703     public void setMonth(int month) {
704         int y = 0;
705         if (month >= 12) {
706             y = month / 12;
707             month %= 12;
708         } else if (month < 0) {
709             y = CalendarUtils.floorDivide(month, 12);
710             month = CalendarUtils.mod(month, 12);
711         }
712         BaseCalendar.Date d = getCalendarDate();
713         if (y != 0) {
714             d.setNormalizedYear(d.getNormalizedYear() + y);
715         }
716         d.setMonth(month + 1); // adjust 0-based to 1-based month numbering
717     }
718
719     /**
720     * Returns the day of the month represented by this Date object.
721     * The value returned is between 1 and 31
722     * representing the day of the month that contains or begins with the
723     * instant in time represented by this Date object, as
724     * interpreted in the local time zone.
725     *
726     * @return the day of the month represented by this date.
727     * @see java.util.Calendar
728     * @deprecated As of JDK version 1.1,
729     * replaced by Calendar.get(Calendar.DAY_OF_MONTH).
730     * @deprecated
731     */
732     @Deprecated
733     public int getDate() {
734         return normalize().getDayOfMonth();
735     }
736
737     /**
738     * Sets the day of the month of this Date object to the
739     * specified value. This Date object is modified so that
740     * it represents a point in time within the specified day of the
741     * month, with the year, month, hour, minute, and second the same
742     * as before, as interpreted in the local time zone. If the date
743     * was April 30, for example, and the date is set to 31, then it
744     * will be treated as if it were on May 1, because April has only
745     * 30 days.
746     *
747     * @param date the day of the month value between 1-31.
```

```
748     * @see      java.util.Calendar
749     * @deprecated As of JDK version 1.1,
750     * replaced by <code>Calendar.set(Calendar.DAY_OF_MONTH, int date)</code>.
751     */
752     @Deprecated
753     public void setDate(int date) {
754         getCalendarDate().setDayOfMonth(date);
755     }
756
757     /**
758     * Returns the day of the week represented by this date. The
759     * returned value (<tt>0</tt> = Sunday, <tt>1</tt> = Monday,
760     * <tt>2</tt> = Tuesday, <tt>3</tt> = Wednesday, <tt>4</tt> =
761     * Thursday, <tt>5</tt> = Friday, <tt>6</tt> = Saturday)
762     * represents the day of the week that contains or begins with
763     * the instant in time represented by this <tt>Date</tt> object,
764     * as interpreted in the local time zone.
765     *
766     * @return     the day of the week represented by this date.
767     * @see      java.util.Calendar
768     * @deprecated As of JDK version 1.1,
769     * replaced by <code>Calendar.get(Calendar.DAY_OF_WEEK)</code>.
770     */
771     @Deprecated
772     public int getDay() {
773         return normalize().getDayOfWeek() - gcal.SUNDAY;
774     }
775
776     /**
777     * Returns the hour represented by this <tt>Date</tt> object. The
778     * returned value is a number (<tt>0</tt> through <tt>23</tt>)
779     * representing the hour within the day that contains or begins
780     * with the instant in time represented by this <tt>Date</tt>
781     * object, as interpreted in the local time zone.
782     *
783     * @return     the hour represented by this date.
784     * @see      java.util.Calendar
785     * @deprecated As of JDK version 1.1,
786     * replaced by <code>Calendar.get(Calendar.HOUR_OF_DAY)</code>.
787     */
788     @Deprecated
789     public int getHours() {
790         return normalize().getHours();
791     }
792
793     /**
794     * Sets the hour of this <tt>Date</tt> object to the specified value.
795     * This <tt>Date</tt> object is modified so that it represents a point
796     * in time within the specified hour of the day, with the year, month,
797     * date, minute, and second the same as before, as interpreted in the
798     * local time zone.
799     *
800     * @param     hours    the hour value.
801     * @see      java.util.Calendar
802     * @denrecated As of JDK version 1.1.
```

```
803     * replaced by Calendar.set(Calendar.HOUR_OF_DAY, int hours).
804     */
805     @Deprecated
806     public void setHours(int hours) {
807         getCalendarDate().setHours(hours);
808     }
809
810     /**
811      * Returns the number of minutes past the hour represented by this date,
812      * as interpreted in the local time zone.
813      * The value returned is between 0 and 59.
814      *
815      * @return the number of minutes past the hour represented by this date.
816      * @see java.util.Calendar
817      * @deprecated As of JDK version 1.1,
818      * replaced by Calendar.get(Calendar.MINUTE).
819      */
820     @Deprecated
821     public int getMinutes() {
822         return normalize().getMinutes();
823     }
824
825     /**
826      * Sets the minutes of this Date object to the specified value.
827      * This Date object is modified so that it represents a point
828      * in time within the specified minute of the hour, with the year, month,
829      * date, hour, and second the same as before, as interpreted in the
830      * local time zone.
831      *
832      * @param minutes the value of the minutes.
833      * @see java.util.Calendar
834      * @deprecated As of JDK version 1.1,
835      * replaced by Calendar.set(Calendar.MINUTE, int minutes).
836      */
837     @Deprecated
838     public void setMinutes(int minutes) {
839         getCalendarDate().setMinutes(minutes);
840     }
841
842     /**
843      * Returns the number of seconds past the minute represented by this date.
844      * The value returned is between 0 and 61. The
845      * values 60 and 61 can only occur on those
846      * Java Virtual Machines that take leap seconds into account.
847      *
848      * @return the number of seconds past the minute represented by this date.
849      * @see java.util.Calendar
850      * @deprecated As of JDK version 1.1,
851      * replaced by Calendar.get(Calendar.SECOND).
852      */
853     @Deprecated
854     public int getSeconds() {
855         return normalize().getSeconds();
856     }
```



```
857
858 /**
859  * Sets the seconds of this <tt>Date</tt> to the specified value.
860  * This <tt>Date</tt> object is modified so that it represents a
861  * point in time within the specified second of the minute, with
862  * the year, month, date, hour, and minute the same as before, as
863  * interpreted in the local time zone.
864  *
865  * @param   seconds   the seconds value.
866  * @see     java.util.Calendar
867  * @deprecated As of JDK version 1.1,
868  *    replaced by <code>Calendar.set(Calendar.SECOND, int seconds)</code>.
869  */
870 @Deprecated
871 public void setSeconds(int seconds) {
872     getCalendarDate().setSeconds(seconds);
873 }
874
875 /**
876  * Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT
877  * represented by this <tt>Date</tt> object.
878  *
879  * @return   the number of milliseconds since January 1, 1970, 00:00:00 GMT
880  *    represented by this date.
881  */
882 public long getTime() {
883     return getTimeImpl();
884 }
885
886 private final long getTimeImpl() {
887     if (cdate != null && !cdate.isNormalized()) {
888         normalize();
889     }
890     return fastTime;
891 }
892
893 /**
894  * Sets this <code>Date</code> object to represent a point in time that is
895  * <code>time</code> milliseconds after January 1, 1970 00:00:00 GMT.
896  *
897  * @param   time       the number of milliseconds.
898  */
899 public void setTime(long time) {
900     fastTime = time;
901     cdate = null;
902 }
903
904 /**
905  * Tests if this date is before the specified date.
906  *
907  * @param   when        a date.
908  * @return   <code>true</code> if and only if the instant of time
909  *    represented by this <tt>Date</tt> object is strictly
910  *    earlier than the instant represented by <tt>when</tt>;
911  *    <code>false</code> otherwise.
```

```

912     * @exception NullPointerException if <code>when</code> is null.
913     */
914     public boolean before(Date when) {
915         return getMillisOf(this) < getMillisOf(when);
916     }
917
918     /**
919     * Tests if this date is after the specified date.
920     *
921     * @param   when    a date.
922     * @return  <code>true</code> if and only if the instant represented
923     *          by this <tt>Date</tt> object is strictly later than the
924     *          instant represented by <tt>when</tt>;
925     *          <code>false</code> otherwise.
926     * @exception NullPointerException if <code>when</code> is null.
927     */
928     public boolean after(Date when) {
929         return getMillisOf(this) > getMillisOf(when);
930     }
931
932     /**
933     * Compares two dates for equality.
934     * The result is <code>true</code> if and only if the argument is
935     * not <code>null</code> and is a <code>Date</code> object that
936     * represents the same point in time, to the millisecond, as this object.
937     * <p>
938     * Thus, two <code>Date</code> objects are equal if and only if the
939     * <code>getTime</code> method returns the same <code>long</code>
940     * value for both.
941     *
942     * @param   obj    the object to compare with.
943     * @return  <code>true</code> if the objects are the same;
944     *          <code>false</code> otherwise.
945     * @see     java.util.Date#getTime()
946     */
947     public boolean equals(Object obj) {
948         return obj instanceof Date && getTime() == ((Date) obj).getTime();
949     }
950
951     /**
952     * Returns the millisecond value of this <code>Date</code> object
953     * without affecting its internal state.
954     */
955     static final long getMillisOf(Date date) {
956         if (date.cdate == null || date.cdate.isNormalized()) {
957             return date.fastTime;
958         }
959         BaseCalendar.Date d = (BaseCalendar.Date) date.cdate.clone();
960         return gcal.getTime(d);
961     }
962
963     /**
964     * Compares two Dates for ordering.
965     *
966     * @param   anotherDate  the <code>Date</code> to be compared

```

```

966     * @param anotherDate the <code>Date</code> to be compared.
967     * @return the value <code>0</code> if the argument Date is equal to
968     *         this Date; a value less than <code>0</code> if this Date
969     *         is before the Date argument; and a value greater than
970     *         <code>0</code> if this Date is after the Date argument.
971     * @since 1.2
972     * @exception NullPointerException if <code>anotherDate</code> is null.
973     */
974     public int compareTo(Date anotherDate) {
975         long thisTime = getMillisOf(this);
976         long anotherTime = getMillisOf(anotherDate);
977         return (thisTime < anotherTime ? -1 : (thisTime == anotherTime ? 0 : 1));
978     }
979
980     /**
981     * Returns a hash code value for this object. The result is the
982     * exclusive OR of the two halves of the primitive <tt>long</tt>
983     * value returned by the {@link Date#getTime}
984     * method. That is, the hash code is the value of the expression:
985     * <blockquote><pre>
986     * (int)(this.getTime()^(this.getTime() >>> 32))</pre></blockquote>
987     *
988     * @return a hash code value for this object.
989     */
990     public int hashCode() {
991         long ht = this.getTime();
992         return (int) ht ^ (int) (ht >> 32);
993     }
994
995     /**
996     * Converts this <code>Date</code> object to a <code>String</code>
997     * of the form:
998     * <blockquote><pre>
999     * dow mon dd hh:mm:ss zzz yyyy</pre></blockquote>
1000     * where:<ul>
1001     * <li><tt>dow</tt> is the day of the week (<tt>Sun, Mon, Tue, Wed,
1002     *     Thu, Fri, Sat</tt>).
1003     * <li><tt>mon</tt> is the month (<tt>Jan, Feb, Mar, Apr, May, Jun,
1004     *     Jul, Aug, Sep, Oct, Nov, Dec</tt>).
1005     * <li><tt>dd</tt> is the day of the month (<tt>01</tt> through
1006     *     <tt>31</tt>), as two decimal digits.
1007     * <li><tt>hh</tt> is the hour of the day (<tt>00</tt> through
1008     *     <tt>23</tt>), as two decimal digits.
1009     * <li><tt>mm</tt> is the minute within the hour (<tt>00</tt> through
1010     *     <tt>59</tt>), as two decimal digits.
1011     * <li><tt>ss</tt> is the second within the minute (<tt>00</tt> through
1012     *     <tt>61</tt>), as two decimal digits.
1013     * <li><tt>zzz</tt> is the time zone (and may reflect daylight saving
1014     *     time). Standard time zone abbreviations include those
1015     *     recognized by the method <tt>parse</tt>. If time zone
1016     *     information is not available, then <tt>zzz</tt> is empty -
1017     *     that is, it consists of no characters at all.
1018     * <li><tt>yyyy</tt> is the year, as four decimal digits.
1019     * </ul>
1020     */

```

```

1021     * @return a string representation of this date.
1022     * @see java.util.Date#toLocaleString()
1023     * @see java.util.Date#toGMTString()
1024     */
1025     public String toString() {
1026         // "EEE MMM dd HH:mm:ss zzz yyyy";
1027         BaseCalendar.Date date = normalize();
1028         StringBuilder sb = new StringBuilder(28);
1029         int index = date.getDayOfWeek();
1030         if (index == gcal.SUNDAY) {
1031             index = 8;
1032         }
1033         convertToAbbr(sb, wtb[index]).append(' '); // EEE
1034         convertToAbbr(sb, wtb[date.getMonth() - 1 + 2 + 7]).append(' '); // MMM
1035         CalendarUtils.printf0d(sb, date.getDayOfMonth(), 2).append(' '); // dd
1036
1037         CalendarUtils.printf0d(sb, date.getHours(), 2).append(':'); // HH
1038         CalendarUtils.printf0d(sb, date.getMinutes(), 2).append(':'); // mm
1039         CalendarUtils.printf0d(sb, date.getSeconds(), 2).append(' '); // ss
1040         TimeZone zi = date.getZone();
1041         if (zi != null) {
1042             sb.append(zi.getDisplayName(date.isDaylightTime(), zi.SHORT, Locale.US)); // zzz
1043         } else {
1044             sb.append("GMT");
1045         }
1046         sb.append(' ').append(date.getYear()); // yyyy
1047         return sb.toString();
1048     }
1049
1050     /**
1051     * Converts the given name to its 3-letter abbreviation (e.g.,
1052     * "monday" -> "Mon") and stored the abbreviation in the given
1053     * <code>StringBuilder</code>.
1054     */
1055     private static final StringBuilder convertToAbbr(StringBuilder sb, String name) {
1056         sb.append(Character.toUpperCase(name.charAt(0)));
1057         sb.append(name.charAt(1)).append(name.charAt(2));
1058         return sb;
1059     }
1060
1061     /**
1062     * Creates a string representation of this <code>Date</code> object in an
1063     * implementation-dependent form. The intent is that the form should
1064     * be familiar to the user of the Java application, wherever it may
1065     * happen to be running. The intent is comparable to that of the
1066     * "<code>%c</code>" format supported by the <code>strftime</code>
1067     * function of ISO&nbsp;C.
1068     *
1069     * @return a string representation of this date, using the locale
1070     *         conventions.
1071     * @see java.text.DateFormat
1072     * @see java.util.Date#toString()
1073     * @see java.util.Date#toGMTString()
1074     * @deprecated As of JDK version 1.1,
1075     *         replaced by <code>DateFormat.format(Date date)</code>.

```

```

1075     * replaced by DateFormatter.format(Date date, Locale locale).
1076     */
1077     @Deprecated
1078     public String toLocaleString() {
1079         DateFormat formatter = DateFormat.getDateTimeInstance();
1080         return formatter.format(this);
1081     }
1082
1083     /**
1084      * Creates a string representation of this Date object of
1085      * the form:
1086      * 

```
d mon yyyy hh:mm:ss GMT
```


1087      * where:
1088      * 


1089      * - d is the day of the month (1 through 31),
1090      *     as one or two decimal digits.

1091      * - mon is the month (Jan, Feb, Mar, Apr, May, Jun, Jul,
1092      *     Aug, Sep, Oct, Nov, Dec).

1093      * - yyyy is the year, as four decimal digits.

1094      * - hh is the hour of the day (00 through 23),
1095      *     as two decimal digits.

1096      * - mm is the minute within the hour (00 through
1097      *     59), as two decimal digits.

1098      * - ss is the second within the minute (00 through
1099      *     61), as two decimal digits.

1100      * - GMT is exactly the ASCII letters "GMT" to indicate
1101      *     Greenwich Mean Time.

1102      * 

1103      * The result does not depend on the local time zone.
1104      *
1105      * @return a string representation of this date, using the Internet GMT
1106      *     conventions.
1107      * @see java.text.DateFormat
1108      * @see java.util.Date#toString()
1109      * @see java.util.Date#toLocaleString()
1110      * @deprecated As of JDK version 1.1,
1111      *     replaced by DateFormatter.format(Date date), using a
1112      *     GMT TimeZone.
1113      */
1114     @Deprecated
1115     public String toGMTString() {
1116         // d MMM yyyy HH:mm:ss 'GMT'
1117         long t = getTime();
1118         BaseCalendar cal = getCalendarSystem(t);
1119         BaseCalendar.Date date =
1120             (BaseCalendar.Date) cal.getCalendarDate(getTime(), (TimeZone)null);
1121         StringBuilder sb = new StringBuilder(32);
1122         CalendarUtils.printf0d(sb, date.getDayOfMonth(), 1).append(' '); // d
1123         CalendarUtils.convertToAbbr(sb, date.getMonth() - 1 + 2 + 7).append(' '); // MMM
1124         sb.append(date.getYear()).append(' '); // yyyy
1125         CalendarUtils.printf0d(sb, date.getHours(), 2).append(':'); // HH
1126         CalendarUtils.printf0d(sb, date.getMinutes(), 2).append(':'); // mm
1127         CalendarUtils.printf0d(sb, date.getSeconds(), 2); // ss
1128         sb.append(" GMT"); // ' GMT'
1129         return sb.toString();

```

```

1130     }
1131
1132     /**
1133      * Returns the offset, measured in minutes, for the local time zone
1134      * relative to UTC that is appropriate for the time represented by
1135      * this Date object.
1136      * <p>
1137      * For example, in Massachusetts, five time zones west of Greenwich:
1138      * <blockquote><pre>
1139      * new Date(96, 1, 14).getTimezoneOffset() returns 300</pre></blockquote>
1140      * because on February 14, 1996, standard time (Eastern Standard Time)
1141      * is in use, which is offset five hours from UTC; but:
1142      * <blockquote><pre>
1143      * new Date(96, 5, 1).getTimezoneOffset() returns 240</pre></blockquote>
1144      * because on June 1, 1996, daylight saving time (Eastern Daylight Time)
1145      * is in use, which is offset only four hours from UTC.<p>
1146      * This method produces the same result as if it computed:
1147      * <blockquote><pre>
1148      * (this.getTime() - UTC(this.getYear(),
1149      *                        this.getMonth(),
1150      *                        this.getDate(),
1151      *                        this.getHours(),
1152      *                        this.getMinutes(),
1153      *                        this.getSeconds())) / (60 * 1000)
1154      * </pre></blockquote>
1155      *
1156      * @return the time-zone offset, in minutes, for the current time zone.
1157      * @see java.util.Calendar#ZONE_OFFSET
1158      * @see java.util.Calendar#DST_OFFSET
1159      * @see java.util.TimeZone#getDefault
1160      * @deprecated As of JDK version 1.1,
1161      * replaced by -(Calendar.get(Calendar.ZONE_OFFSET) +
1162      * Calendar.get(Calendar.DST_OFFSET)) / (60 * 1000).
1163      */
1164     @Deprecated
1165     public int getTimezoneOffset() {
1166         int zoneOffset;
1167         if (cdate == null) {
1168             TimeZone tz = TimeZone.getDefaultRef();
1169             if (tz instanceof ZoneInfo) {
1170                 zoneOffset = ((ZoneInfo)tz).getOffsets(fastTime, null);
1171             } else {
1172                 zoneOffset = tz.getOffset(fastTime);
1173             }
1174         } else {
1175             normalize();
1176             zoneOffset = cdate.getZoneOffset();
1177         }
1178         return -zoneOffset/60000; // convert to minutes
1179     }
1180
1181     private final BaseCalendar.Date getCalendarDate() {
1182         if (cdate == null) {
1183             BaseCalendar cal = getCalendarSystem(fastTime);
1184             cdate = (BaseCalendar.Date) cal.getCalendarDate(fastTime,

```

```
1185                                     TimeZone.getDefaultRef());
1186     }
1187     return cdate;
1188 }
1189
1190 private final BaseCalendar.Date normalize() {
1191     if (cdate == null) {
1192         BaseCalendar cal = getCalendarSystem(fastTime);
1193         cdate = (BaseCalendar.Date) cal.getCalendarDate(fastTime,
1194                                                         TimeZone.getDefaultRef());
1195         return cdate;
1196     }
1197
1198     // Normalize cdate with the TimeZone in cdate first. This is
1199     // required for the compatible behavior.
1200     if (!cdate.isNormalized()) {
1201         cdate = normalize(cdate);
1202     }
1203
1204     // If the default TimeZone has changed, then recalculate the
1205     // fields with the new TimeZone.
1206     TimeZone tz = TimeZone.getDefaultRef();
1207     if (tz != cdate.getTimeZone()) {
1208         cdate.setTimeZone(tz);
1209         CalendarSystem cal = getCalendarSystem(cdate);
1210         cal.getCalendarDate(fastTime, cdate);
1211     }
1212     return cdate;
1213 }
1214
1215 // fastTime and the returned data are in sync upon return.
1216 private final BaseCalendar.Date normalize(BaseCalendar.Date date) {
1217     int y = date.getNormalizedYear();
1218     int m = date.getMonth();
1219     int d = date.getDayOfMonth();
1220     int hh = date.getHours();
1221     int mm = date.getMinutes();
1222     int ss = date.getSeconds();
1223     int ms = date.getMillis();
1224     TimeZone tz = date.getTimeZone();
1225
1226     // If the specified year can't be handled using a long value
1227     // in milliseconds, GregorianCalendar is used for full
1228     // compatibility with underflow and overflow. This is required
1229     // by some JCK tests. The limits are based max year values -
1230     // years that can be represented by max values of d, hh, mm,
1231     // ss and ms. Also, let GregorianCalendar handle the default
1232     // cutover year so that we don't need to worry about the
1233     // transition here.
1234     if (y == 1582 || y > 2800000000 || y < -2800000000) {
1235         if (tz == null) {
1236             tz = TimeZone.getTimeZone("GMT");
1237         }
1238         GregorianCalendar gc = new GregorianCalendar(tz);
1239         gc.set(1970, 0, 1, 0, 0, 0);
1240         gc.add(Calendar.YEAR, y);
1241         return gc.getTime();
1242     }
1243     return date;
1244 }
```

```

1239         gc.clear();
1240         gc.set(gc.MILLISECOND, ms);
1241         gc.set(y, m-1, d, hh, mm, ss);
1242         fastTime = gc.getTimeInMillis();
1243         BaseCalendar cal = getCalendarSystem(fastTime);
1244         date = (BaseCalendar.Date) cal.getCalendarDate(fastTime, tz);
1245         return date;
1246     }
1247
1248     BaseCalendar cal = getCalendarSystem(y);
1249     if (cal != getCalendarSystem(date)) {
1250         date = (BaseCalendar.Date) cal.newCalendarDate(tz);
1251         date.setNormalizedDate(y, m, d).setTimeOfDay(hh, mm, ss, ms);
1252     }
1253     // Perform the GregorianCalendar-style normalization.
1254     fastTime = cal.getTime(date);
1255
1256     // In case the normalized date requires the other calendar
1257     // system, we need to recalculate it using the other one.
1258     BaseCalendar ncal = getCalendarSystem(fastTime);
1259     if (ncal != cal) {
1260         date = (BaseCalendar.Date) ncal.newCalendarDate(tz);
1261         date.setNormalizedDate(y, m, d).setTimeOfDay(hh, mm, ss, ms);
1262         fastTime = ncal.getTime(date);
1263     }
1264     return date;
1265 }
1266
1267 /**
1268  * Returns the Gregorian or Julian calendar system to use with the
1269  * given date. Use Gregorian from October 15, 1582.
1270  *
1271  * @param year normalized calendar year (not -1900)
1272  * @return the CalendarSystem to use for the specified date
1273  */
1274 private static final BaseCalendar getCalendarSystem(int year) {
1275     if (year >= 1582) {
1276         return gcal;
1277     }
1278     return getJulianCalendar();
1279 }
1280
1281 private static final BaseCalendar getCalendarSystem(long utc) {
1282     // Quickly check if the time stamp given by `utc' is the Epoch
1283     // or later. If it's before 1970, we convert the cutover to
1284     // local time to compare.
1285     if (utc >= 0
1286         || utc >= GregorianCalendar.DEFAULT_GREGORIAN_CUTOVER
1287             - TimeZone.getDefaultRef().getOffset(utc)) {
1288         return gcal;
1289     }
1290     return getJulianCalendar();
1291 }
1292
1293 private static final BaseCalendar getCalendarSystem(BaseCalendar.Date cdate) {

```



```
1294         if (jcal == null) {
1295             return gcal;
1296         }
1297         if (cdate.getEra() != null) {
1298             return jcal;
1299         }
1300         return gcal;
1301     }
1302
1303     synchronized private static final BaseCalendar getJulianCalendar() {
1304         if (jcal == null) {
1305             jcal = (BaseCalendar) CalendarSystem.forName("julian");
1306         }
1307         return jcal;
1308     }
1309
1310     /**
1311      * Save the state of this object to a stream (i.e., serialize it).
1312      *
1313      * @serialData The value returned by <code>getTime()</code>
1314      *              is emitted (long). This represents the offset from
1315      *              January 1, 1970, 00:00:00 GMT in milliseconds.
1316      */
1317     private void writeObject(ObjectOutputStream s)
1318         throws IOException
1319     {
1320         s.writeLong(getTimeImpl());
1321     }
1322
1323     /**
1324      * Reconstitute this object from a stream (i.e., deserialize it).
1325      */
1326     private void readObject(ObjectInputStream s)
1327         throws IOException, ClassNotFoundException
1328     {
1329         fastTime = s.readLong();
1330     }
1331 }
```