```
/*
 * Copyright (c) 1994, 2010, Oracle and/or its affiliates. All rights reserved.
 * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS FILE HEADER.
 *
 * This code is free software; you can redistribute it and/or modify it
 * under the terms of the GNU General Public License version 2 only, as
 * published by the Free Software Foundation.  Oracle designates this
 * particular file as subject to the "Classpath" exception as provided
 * by Oracle in the LICENSE file that accompanied this code.
 *
 * This code is distributed in the hope that it will be useful, but WITHOUT
 * ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
 * FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
 * version 2 for more details (a copy is included in the LICENSE file that
 * accompanied this code).
 *
 * You should have received a copy of the GNU General Public License version
 * 2 along with this work; if not, write to the Free Software Foundation,
 * Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.
 *
 * Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA
 * or visit www.oracle.com if you need additional information or have any
 * questions.
 */

package java.util;

import java.text.DateFormat;
import java.io.IOException;
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.lang.ref.SoftReference;
import sun.util.calendar.BaseCalendar;
import sun.util.calendar.CalendarDate;
import sun.util.calendar.CalendarSystem;
import sun.util.calendar.CalendarUtils;
import sun.util.calendar.Era;
```

```
import sun.util.calendar.Gregorian;
import sun.util.calendar.ZoneInfo;

/**
 * The class <code>Date</code> represents a specific instant
 * in time, with millisecond precision.
 * <p>
 * Prior to JDK 1.1, the class <code>Date</code> had two additional
 * functions.  It allowed the interpretation of dates as year, month, day, hour,
 * minute, and second values.  It also allowed the formatting and parsing
 * of date strings.  Unfortunately, the API for these functions was not
 * amenable to internationalization.  As of JDK 1.1, the
 * <code>Calendar</code> class should be used to convert between dates and time
 * fields and the <code>DateFormat</code> class should be used to format and
 * parse date strings.
 * The corresponding methods in <code>Date</code> are deprecated.
 * <p>
 * Although the <code>Date</code> class is intended to reflect
 * coordinated universal time (UTC), it may not do so exactly,
 * depending on the host environment of the Java Virtual Machine.
 * Nearly all modern operating systems assume that 1 day =
 * 24 &times; 60 &times; 60 = 86400 seconds
 * in all cases. In UTC, however, about once every year or two there
 * is an extra second, called a "leap second." The leap
 * second is always added as the last second of the day, and always
 * on December 31 or June 30. For example, the last minute of the
 * year 1995 was 61 seconds long, thanks to an added leap second.
 * Most computer clocks are not accurate enough to be able to reflect
 * the leap-second distinction.
 * <p>
 * Some computer standards are defined in terms of Greenwich mean
 * time (GMT), which is equivalent to universal time (UT).  GMT is
 * the "civil" name for the standard; UT is the
 * "scientific" name for the same standard. The
 * distinction between UTC and UT is that UTC is based on an atomic
 * clock and UT is based on astronomical observations, which for all
 * practical purposes is an invisibly fine hair to split. Because the
 * earth's rotation is not uniform (it slows down and speeds up
 * in complicated ways), UT does not always flow uniformly. Leap
 * seconds are introduced as needed into UTC so as to keep UTC within
 * 0.9 seconds of UT1, which is a version of UT with certain
 * corrections applied. There are other time and date systems as
 * well; for example, the time scale used by the satellite-based
 * global positioning system (GPS) is synchronized to UTC but is
 * <i>not</i> adjusted for leap seconds. An interesting source of
 * further information is the U.S. Naval Observatory, particularly
 * the Directorate of Time at:
 * <blockquote><pre>
 *     <a href=http://tycho.usno.navy.mil>http://tycho.usno.navy.mil</a>
 * </pre></blockquote>
 * <p>
 * and their definitions of "Systems of Time" at:
 * <blockquote><pre>
 *     <a href=http://tycho.usno.navy.mil/systime.html>http://tycho.usno.navy.mil/systime.html<
 * </pre></blockquote>
```

```
 * <p>
 * In all methods of class <code>Date</code> that accept or return
 * year, month, date, hours, minutes, and seconds values, the
 * following representations are used:
 * <ul>
 * <li>A year <i>y</i> is represented by the integer
 *     <i>y</i> <code>- 1900</code>.
 * <li>A month is represented by an integer from 0 to 11; 0 is January,
 *     1 is February, and so forth; thus 11 is December.
 * <li>A date (day of month) is represented by an integer from 1 to 31
 *     in the usual manner.
 * <li>An hour is represented by an integer from 0 to 23. Thus, the hour
 *     from midnight to 1 a.m. is hour 0, and the hour from noon to 1
 *     p.m. is hour 12.
 * <li>A minute is represented by an integer from 0 to 59 in the usual manner.
 * <li>A second is represented by an integer from 0 to 61; the values 60 and
 *     61 occur only for leap seconds and even then only in Java
 *     implementations that actually track leap seconds correctly. Because
 *     of the manner in which leap seconds are currently introduced, it is
 *     extremely unlikely that two leap seconds will occur in the same
 *     minute, but this specification follows the date and time conventions
 *     for ISO C.
 * </ul>
 * <p>
 * In all cases, arguments given to methods for these purposes need
 * not fall within the indicated ranges; for example, a date may be
 * specified as January 32 and is interpreted as meaning February 1.
 *
 * @author  James Gosling
 * @author  Arthur van Hoff
 * @author  Alan Liu
 * @see     java.text.DateFormat
 * @see     java.util.Calendar
 * @see     java.util.TimeZone
 * @since   JDK1.0
 */
public class Date
    implements java.io.Serializable, Cloneable, Comparable<Date>
{
    private static final BaseCalendar gcal =
                                CalendarSystem.getGregorianCalendar();
    private static BaseCalendar jcal;

    private transient long fastTime;

    /*
     * If cdate is null, then fastTime indicates the time in millis.
     * If cdate.isNormalized() is true, then fastTime and cdate are in
     * synch. Otherwise, fastTime is ignored, and cdate indicates the
     * time.
     */
    private transient BaseCalendar.Date cdate;

    // Initialized just before the value is used. See parse().
    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
```

```
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>,
```

```
     * <code>hrs</code>, and <code>min</code> arguments, in the local
     * time zone.
     *
     * @param   year    the year minus 1900.
     * @param   month   the month between 0-11.
     * @param   date    the day of the month between 1-31.
     * @param   hrs     the hours between 0-23.
     * @param   min     the minutes between 0-59.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.set(year + 1900, month, date,
     * hrs, min)</code> or <code>GregorianCalendar(year + 1900,
     * month, date, hrs, min)</code>.
     */
    @Deprecated
    public Date(int year, int month, int date, int hrs, int min) {
        this(year, month, date, hrs, min, 0);
    }

    /**
     * Allocates a <code>Date</code> object and initializes it so that
     * it represents the instant at the start of the second specified
     * by the <code>year</code>, <code>month</code>, <code>date</code>,
     * <code>hrs</code>, <code>min</code>, and <code>sec</code> arguments,
     * in the local time zone.
     *
     * @param   year    the year minus 1900.
     * @param   month   the month between 0-11.
     * @param   date    the day of the month between 1-31.
     * @param   hrs     the hours between 0-23.
     * @param   min     the minutes between 0-59.
     * @param   sec     the seconds between 0-59.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.set(year + 1900, month, date,
     * hrs, min, sec)</code> or <code>GregorianCalendar(year + 1900,
     * month, date, hrs, min, sec)</code>.
     */
    @Deprecated
    public Date(int year, int month, int date, int hrs, int min, int sec) {
        int y = year + 1900;
        // month is 0-based. So we have to normalize month to support Long.MAX_VALUE.
        if (month >= 12) {
            y += month / 12;
            month %= 12;
        } else if (month < 0) {
            y += CalendarUtils.floorDivide(month, 12);
            month = CalendarUtils.mod(month, 12);
        }
        BaseCalendar cal = getCalendarSystem(y);
        cdate = (BaseCalendar.Date) cal.newCalendarDate(TimeZone.getDefaultRef());
        cdate.setNormalizedDate(y, month + 1, date).setTimeOfDay(hrs, min, sec, 0);
        getTimeImpl();
        cdate = null;
    }
```

```
    }

    /**
     * Allocates a <code>Date</code> object and initializes it so that
     * it represents the date and time indicated by the string
     * <code>s</code>, which is interpreted as if by the
     * {@link Date#parse} method.
     *
     * @param   s   a string representation of the date.
     * @see     java.text.DateFormat
     * @see     java.util.Date#parse(java.lang.String)
     * @deprecated As of JDK version 1.1,
     * replaced by <code>DateFormat.parse(String s)</code>.
     */
    @Deprecated
    public Date(String s) {
        this(parse(s));
    }

    /**
     * Return a copy of this object.
     */
    public Object clone() {
        Date d = null;
        try {
            d = (Date)super.clone();
            if (cdate != null) {
                d.cdate = (BaseCalendar.Date) cdate.clone();
            }
        } catch (CloneNotSupportedException e) {} // Won't happen
        return d;
    }

    /**
     * Determines the date and time based on the arguments. The
     * arguments are interpreted as a year, month, day of the month,
     * hour of the day, minute within the hour, and second within the
     * minute, exactly as for the <tt>Date</tt> constructor with six
     * arguments, except that the arguments are interpreted relative
     * to UTC rather than to the local time zone. The time indicated is
     * returned represented as the distance, measured in milliseconds,
     * of that time from the epoch (00:00:00 GMT on January 1, 1970).
     *
     * @param   year    the year minus 1900.
     * @param   month   the month between 0-11.
     * @param   date    the day of the month between 1-31.
     * @param   hrs     the hours between 0-23.
     * @param   min     the minutes between 0-59.
     * @param   sec     the seconds between 0-59.
     * @return  the number of milliseconds since January 1, 1970, 00:00:00 GMT for
     *          the date and time specified by the arguments.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.set(year + 1900, month, date,
     * hrs, min, sec)</code> or <code>GregorianCalendar(year + 1900,
```

```
     * month, date, hrs, min, sec)</code>, using a UTC
     * <code>TimeZone</code>, followed by <code>Calendar.getTime().getTime()</code>.
     */
    @Deprecated
    public static long UTC(int year, int month, int date,
                           int hrs, int min, int sec) {
        int y = year + 1900;
        // month is 0-based. So we have to normalize month to support Long.MAX_VALUE.
        if (month >= 12) {
            y += month / 12;
            month %= 12;
        } else if (month < 0) {
            y += CalendarUtils.floorDivide(month, 12);
            month = CalendarUtils.mod(month, 12);
        }
        int m = month + 1;
        BaseCalendar cal = getCalendarSystem(y);
        BaseCalendar.Date udate = (BaseCalendar.Date) cal.newCalendarDate(null);
        udate.setNormalizedDate(y, m, date).setTimeOfDay(hrs, min, sec, 0);

        // Use a Date instance to perform normalization. Its fastTime
        // is the UTC value after the normalization.
        Date d = new Date(0);
        d.normalize(udate);
        return d.fastTime;
    }

    /**
     * Attempts to interpret the string <tt>s</tt> as a representation
     * of a date and time. If the attempt is successful, the time
     * indicated is returned represented as the distance, measured in
     * milliseconds, of that time from the epoch (00:00:00 GMT on
     * January 1, 1970). If the attempt fails, an
     * <tt>IllegalArgumentException</tt> is thrown.
     * <p>
     * It accepts many syntaxes; in particular, it recognizes the IETF
     * standard date syntax: "Sat, 12 Aug 1995 13:30:00 GMT". It also
     * understands the continental U.S. time-zone abbreviations, but for
     * general use, a time-zone offset should be used: "Sat, 12 Aug 1995
     * 13:30:00 GMT+0430" (4 hours, 30 minutes west of the Greenwich
     * meridian). If no time zone is specified, the local time zone is
     * assumed. GMT and UTC are considered equivalent.
     * <p>
     * The string <tt>s</tt> is processed from left to right, looking for
     * data of interest. Any material in <tt>s</tt> that is within the
     * ASCII parenthesis characters <tt>(</tt> and <tt>)</tt> is ignored.
     * Parentheses may be nested. Otherwise, the only characters permitted
     * within <tt>s</tt> are these ASCII characters:
     * <blockquote><pre>
     * abcdefghijklmnopqrstuvwxyz
     * ABCDEFGHIJKLMNOPQRSTUVWXYZ
     * 0123456789,+-:/</pre></blockquote>
     * and whitespace characters.<p>
     * A consecutive sequence of decimal digits is treated as a decimal
     * number:<ul>
```

```
         * <li>If a number is preceded by <tt>+</tt> or <tt>-</tt> and a year
         *     has already been recognized, then the number is a time-zone
         *     offset. If the number is less than 24, it is an offset measured
         *     in hours. Otherwise, it is regarded as an offset in minutes,
         *     expressed in 24-hour time format without punctuation. A
         *     preceding <tt>-</tt> means a westward offset. Time zone offsets
         *     are always relative to UTC (Greenwich). Thus, for example,
         *     <tt>-5</tt> occurring in the string would mean "five hours west
         *     of Greenwich" and <tt>+0430</tt> would mean "four hours and
         *     thirty minutes east of Greenwich." It is permitted for the
         *     string to specify <tt>GMT</tt>, <tt>UT</tt>, or <tt>UTC</tt>
         *     redundantly-for example, <tt>GMT-5</tt> or <tt>utc+0430</tt>.
         * <li>The number is regarded as a year number if one of the
         *     following conditions is true:
         * <ul>
         *     <li>The number is equal to or greater than 70 and followed by a
         *         space, comma, slash, or end of string
         *     <li>The number is less than 70, and both a month and a day of
         *         the month have already been recognized</li>
         * </ul>
         *     If the recognized year number is less than 100, it is
         *     interpreted as an abbreviated year relative to a century of
         *     which dates are within 80 years before and 19 years after
         *     the time when the Date class is initialized.
         *     After adjusting the year number, 1900 is subtracted from
         *     it. For example, if the current year is 1999 then years in
         *     the range 19 to 99 are assumed to mean 1919 to 1999, while
         *     years from 0 to 18 are assumed to mean 2000 to 2018.  Note
         *     that this is slightly different from the interpretation of
         *     years less than 100 that is used in {@link java.text.SimpleDateFormat}.
         * <li>If the number is followed by a colon, it is regarded as an hour,
         *     unless an hour has already been recognized, in which case it is
         *     regarded as a minute.
         * <li>If the number is followed by a slash, it is regarded as a month
         *     (it is decreased by 1 to produce a number in the range <tt>0</tt>
         *     to <tt>11</tt>), unless a month has already been recognized, in
         *     which case it is regarded as a day of the month.
         * <li>If the number is followed by whitespace, a comma, a hyphen, or
         *     end of string, then if an hour has been recognized but not a
         *     minute, it is regarded as a minute; otherwise, if a minute has
         *     been recognized but not a second, it is regarded as a second;
         *     otherwise, it is regarded as a day of the month. </ul><p>
         * A consecutive sequence of letters is regarded as a word and treated
         * as follows:<ul>
         * <li>A word that matches <tt>AM</tt>, ignoring case, is ignored (but
         *     the parse fails if an hour has not been recognized or is less
         *     than <tt>1</tt> or greater than <tt>12</tt>).
         * <li>A word that matches <tt>PM</tt>, ignoring case, adds <tt>12</tt>
         *     to the hour (but the parse fails if an hour has not been
         *     recognized or is less than <tt>1</tt> or greater than <tt>12</tt>).
         * <li>Any word that matches any prefix of <tt>SUNDAY, MONDAY, TUESDAY,
         *     WEDNESDAY, THURSDAY, FRIDAY</tt>, or <tt>SATURDAY</tt>, ignoring
         *     case, is ignored. For example, <tt>sat, Friday, TUE</tt>, and
         *     <tt>Thurs</tt> are ignored.
         * <li>Otherwise, any word that matches any prefix of <tt>JANUARY
```

```
     * <li>Otherwise, any word that matches any prefix of <tt>JANUARY,
     *     FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER,
     *     OCTOBER, NOVEMBER</tt>, or <tt>DECEMBER</tt>, ignoring case, and
     *     considering them in the order given here, is recognized as
     *     specifying a month and is converted to a number (<tt>0</tt> to
     *     <tt>11</tt>). For example, <tt>aug, Sept, april</tt>, and
     *     <tt>NOV</tt> are recognized as months. So is <tt>Ma</tt>, which
     *     is recognized as <tt>MARCH</tt>, not <tt>MAY</tt>.
     * <li>Any word that matches <tt>GMT, UT</tt>, or <tt>UTC</tt>, ignoring
     *     case, is treated as referring to UTC.
     * <li>Any word that matches <tt>EST, CST, MST</tt>, or <tt>PST</tt>,
     *     ignoring case, is recognized as referring to the time zone in
     *     North America that is five, six, seven, or eight hours west of
     *     Greenwich, respectively. Any word that matches <tt>EDT, CDT,
     *     MDT</tt>, or <tt>PDT</tt>, ignoring case, is recognized as
     *     referring to the same time zone, respectively, during daylight
     *     saving time.</ul><p>
     * Once the entire string s has been scanned, it is converted to a time
     * result in one of two ways. If a time zone or time-zone offset has been
     * recognized, then the year, month, day of month, hour, minute, and
     * second are interpreted in UTC and then the time-zone offset is
     * applied. Otherwise, the year, month, day of month, hour, minute, and
     * second are interpreted in the local time zone.
     *
     * @param   s   a string to be parsed as a date.
     * @return  the number of milliseconds since January 1, 1970, 00:00:00 GMT
     *          represented by the string argument.
     * @see     java.text.DateFormat
     * @deprecated As of JDK version 1.1,
     * replaced by <code>DateFormat.parse(String s)</code>.
     */
    @Deprecated
    public static long parse(String s) {
        int year = Integer.MIN_VALUE;
        int mon = -1;
        int mday = -1;
        int hour = -1;
        int min = -1;
        int sec = -1;
        int millis = -1;
        int c = -1;
        int i = 0;
        int n = -1;
        int wst = -1;
        int tzoffset = -1;
        int prevc = 0;
    syntax:
        {
            if (s == null)
                break syntax;
            int limit = s.length();
            while (i < limit) {
                c = s.charAt(i);
                i++;
                if (c <= ' ' || c == ',')
```

```
                    continue;
                if (c == '(') { // skip comments
                    int depth = 1;
                    while (i < limit) {
                        c = s.charAt(i);
                        i++;
                        if (c == '(') depth++;
                        else if (c == ')')
                            if (--depth <= 0)
                                break;
                    }
                    continue;
                }
                if ('0' <= c && c <= '9') {
                    n = c - '0';
                    while (i < limit && '0' <= (c = s.charAt(i)) && c <= '9') {
                        n = n * 10 + c - '0';
                        i++;
                    }
                    if (prevc == '+' || prevc == '-' && year != Integer.MIN_VALUE) {
                        // timezone offset
                        if (n < 24)
                            n = n * 60; // EG. "GMT-3"
                        else
                            n = n % 100 + n / 100 * 60; // eg "GMT-0430"
                        if (prevc == '+')   // plus means east of GMT
                            n = -n;
                        if (tzoffset != 0 && tzoffset != -1)
                            break syntax;
                        tzoffset = n;
                    } else if (n >= 70)
                        if (year != Integer.MIN_VALUE)
                            break syntax;
                        else if (c <= ' ' || c == ',' || c == '/' || i >= limit)
                            // year = n < 1900 ? n : n - 1900;
                            year = n;
                        else
                            break syntax;
                    else if (c == ':')
                        if (hour < 0)
                            hour = (byte) n;
                        else if (min < 0)
                            min = (byte) n;
                        else
                            break syntax;
                    else if (c == '/')
                        if (mon < 0)
                            mon = (byte) (n - 1);
                        else if (mday < 0)
                            mday = (byte) n;
                        else
                            break syntax;
                    else if (i < limit && c != ',' && c > ' ' && c != '-')
                        break syntax;
                    else if (hour >= 0 && min < 0)
```

```
                    else if (hour >= 0 && min < 0)
                        min = (byte) n;
                    else if (min >= 0 && sec < 0)
                        sec = (byte) n;
                    else if (mday < 0)
                        mday = (byte) n;
                    // Handle two-digit years < 70 (70-99 handled above).
                    else if (year == Integer.MIN_VALUE && mon >= 0 && mday >= 0)
                        year = n;
                    else
                        break syntax;
                    prevc = 0;
                } else if (c == '/' || c == ':' || c == '+' || c == '-')
                    prevc = c;
                else {
                    int st = i - 1;
                    while (i < limit) {
                        c = s.charAt(i);
                        if (!('A' <= c && c <= 'Z' || 'a' <= c && c <= 'z'))
                            break;
                        i++;
                    }
                    if (i <= st + 1)
                        break syntax;
                    int k;
                    for (k = wtb.length; --k >= 0;)
                        if (wtb[k].regionMatches(true, 0, s, st, i - st)) {
                            int action = ttb[k];
                            if (action != 0) {
                                if (action == 1) {  // pm
                                    if (hour > 12 || hour < 1)
                                        break syntax;
                                    else if (hour < 12)
                                        hour += 12;
                                } else if (action == 14) {  // am
                                    if (hour > 12 || hour < 1)
                                        break syntax;
                                    else if (hour == 12)
                                        hour = 0;
                                } else if (action <= 13) {  // month!
                                    if (mon < 0)
                                        mon = (byte) (action - 2);
                                    else
                                        break syntax;
                                } else {
                                    tzoffset = action - 10000;
                                }
                            }
                            break;
                        }
                    if (k < 0)
                        break syntax;
                    prevc = 0;
                }
            }
```

```
                if (year == Integer.MIN_VALUE || mon < 0 || mday < 0)
                    break syntax;
                // Parse 2-digit years within the correct default century.
                if (year < 100) {
                    synchronized (Date.class) {
                        if (defaultCenturyStart == 0) {
                            defaultCenturyStart = gcal.getCalendarDate().getYear() - 80;
                        }
                    }
                    year += (defaultCenturyStart / 100) * 100;
                    if (year < defaultCenturyStart) year += 100;
                }
                if (sec < 0)
                    sec = 0;
                if (min < 0)
                    min = 0;
                if (hour < 0)
                    hour = 0;
                BaseCalendar cal = getCalendarSystem(year);
                if (tzoffset == -1)  { // no time zone specified, have to use local
                    BaseCalendar.Date ldate = (BaseCalendar.Date) cal.newCalendarDate(TimeZone.getD
                    ldate.setDate(year, mon + 1, mday);
                    ldate.setTimeOfDay(hour, min, sec, 0);
                    return cal.getTime(ldate);
                }
                BaseCalendar.Date udate = (BaseCalendar.Date) cal.newCalendarDate(null); // no time
                udate.setDate(year, mon + 1, mday);
                udate.setTimeOfDay(hour, min, sec, 0);
                return cal.getTime(udate) + tzoffset * (60 * 1000);
            }
            // syntax error
            throw new IllegalArgumentException();
        }
        private final static String wtb[] = {
            "am", "pm",
            "monday", "tuesday", "wednesday", "thursday", "friday",
            "saturday", "sunday",
            "january", "february", "march", "april", "may", "june",
            "july", "august", "september", "october", "november", "december",
            "gmt", "ut", "utc", "est", "edt", "cst", "cdt",
            "mst", "mdt", "pst", "pdt"
        };
        private final static int ttb[] = {
            14, 1, 0, 0, 0, 0, 0, 0, 0,
            2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
            10000 + 0, 10000 + 0, 10000 + 0,    // GMT/UT/UTC
            10000 + 5 * 60, 10000 + 4 * 60,     // EST/EDT
            10000 + 6 * 60, 10000 + 5 * 60,     // CST/CDT
            10000 + 7 * 60, 10000 + 6 * 60,     // MST/MDT
            10000 + 8 * 60, 10000 + 7 * 60      // PST/PDT
        };

        /**
         * Returns a value that is the result of subtracting 1900 from the
         * year that contains or begins with the instant in time represented
```

```
     * by this <code>Date</code> object, as interpreted in the local
     * time zone.
     *
     * @return  the year represented by this date, minus 1900.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.get(Calendar.YEAR) - 1900</code>.
     */
    @Deprecated
    public int getYear() {
        return normalize().getYear() - 1900;
    }

    /**
     * Sets the year of this <tt>Date</tt> object to be the specified
     * value plus 1900. This <code>Date</code> object is modified so
     * that it represents a point in time within the specified year,
     * with the month, date, hour, minute, and second the same as
     * before, as interpreted in the local time zone. (Of course, if
     * the date was February 29, for example, and the year is set to a
     * non-leap year, then the new date will be treated as if it were
     * on March 1.)
     *
     * @param   year    the year value.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.set(Calendar.YEAR, year + 1900)</code>.
     */
    @Deprecated
    public void setYear(int year) {
        getCalendarDate().setNormalizedYear(year + 1900);
    }

    /**
     * Returns a number representing the month that contains or begins
     * with the instant in time represented by this <tt>Date</tt> object.
     * The value returned is between <code>0</code> and <code>11</code>,
     * with the value <code>0</code> representing January.
     *
     * @return  the month represented by this date.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.get(Calendar.MONTH)</code>.
     */
    @Deprecated
    public int getMonth() {
        return normalize().getMonth() - 1; // adjust 1-based to 0-based
    }

    /**
     * Sets the month of this date to the specified value. This
     * <tt>Date</tt> object is modified so that it represents a point
     * in time within the specified month, with the year, date, hour,
     * minute, and second the same as before, as interpreted in the
     * local time zone. If the date was October 31, for example, and
```

```
     * local time zone. If the date was October 31, for example, and
     * the month is set to June, then the new date will be treated as
     * if it were on July 1, because June has only 30 days.
     *
     * @param   month   the month value between 0-11.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.set(Calendar.MONTH, int month)</code>.
     */
    @Deprecated
    public void setMonth(int month) {
        int y = 0;
        if (month >= 12) {
            y = month / 12;
            month %= 12;
        } else if (month < 0) {
            y = CalendarUtils.floorDivide(month, 12);
            month = CalendarUtils.mod(month, 12);
        }
        BaseCalendar.Date d = getCalendarDate();
        if (y != 0) {
            d.setNormalizedYear(d.getNormalizedYear() + y);
        }
        d.setMonth(month + 1); // adjust 0-based to 1-based month numbering
    }

    /**
     * Returns the day of the month represented by this <tt>Date</tt> object.
     * The value returned is between <code>1</code> and <code>31</code>
     * representing the day of the month that contains or begins with the
     * instant in time represented by this <tt>Date</tt> object, as
     * interpreted in the local time zone.
     *
     * @return  the day of the month represented by this date.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.get(Calendar.DAY_OF_MONTH)</code>.
     * @deprecated
     */
    @Deprecated
    public int getDate() {
        return normalize().getDayOfMonth();
    }

    /**
     * Sets the day of the month of this <tt>Date</tt> object to the
     * specified value. This <tt>Date</tt> object is modified so that
     * it represents a point in time within the specified day of the
     * month, with the year, month, hour, minute, and second the same
     * as before, as interpreted in the local time zone. If the date
     * was April 30, for example, and the date is set to 31, then it
     * will be treated as if it were on May 1, because April has only
     * 30 days.
     *
     * @param   date   the day of the month value between 1-31.
```

```
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.set(Calendar.DAY_OF_MONTH, int date)</code>.
     */
    @Deprecated
    public void setDate(int date) {
        getCalendarDate().setDayOfMonth(date);
    }

    /**
     * Returns the day of the week represented by this date. The
     * returned value (<tt>0</tt> = Sunday, <tt>1</tt> = Monday,
     * <tt>2</tt> = Tuesday, <tt>3</tt> = Wednesday, <tt>4</tt> =
     * Thursday, <tt>5</tt> = Friday, <tt>6</tt> = Saturday)
     * represents the day of the week that contains or begins with
     * the instant in time represented by this <tt>Date</tt> object,
     * as interpreted in the local time zone.
     *
     * @return  the day of the week represented by this date.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.get(Calendar.DAY_OF_WEEK)</code>.
     */
    @Deprecated
    public int getDay() {
        return normalize().getDayOfWeek() - gcal.SUNDAY;
    }

    /**
     * Returns the hour represented by this <tt>Date</tt> object. The
     * returned value is a number (<tt>0</tt> through <tt>23</tt>)
     * representing the hour within the day that contains or begins
     * with the instant in time represented by this <tt>Date</tt>
     * object, as interpreted in the local time zone.
     *
     * @return  the hour represented by this date.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.get(Calendar.HOUR_OF_DAY)</code>.
     */
    @Deprecated
    public int getHours() {
        return normalize().getHours();
    }

    /**
     * Sets the hour of this <tt>Date</tt> object to the specified value.
     * This <tt>Date</tt> object is modified so that it represents a point
     * in time within the specified hour of the day, with the year, month,
     * date, minute, and second the same as before, as interpreted in the
     * local time zone.
     *
     * @param   hours   the hour value.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
```

```
     * replaced by <code>Calendar.set(Calendar.HOUR_OF_DAY, int hours)</code>.
     */
    @Deprecated
    public void setHours(int hours) {
        getCalendarDate().setHours(hours);
    }

    /**
     * Returns the number of minutes past the hour represented by this date,
     * as interpreted in the local time zone.
     * The value returned is between <code>0</code> and <code>59</code>.
     *
     * @return  the number of minutes past the hour represented by this date.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.get(Calendar.MINUTE)</code>.
     */
    @Deprecated
    public int getMinutes() {
        return normalize().getMinutes();
    }

    /**
     * Sets the minutes of this <tt>Date</tt> object to the specified value.
     * This <tt>Date</tt> object is modified so that it represents a point
     * in time within the specified minute of the hour, with the year, month,
     * date, hour, and second the same as before, as interpreted in the
     * local time zone.
     *
     * @param   minutes   the value of the minutes.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.set(Calendar.MINUTE, int minutes)</code>.
     */
    @Deprecated
    public void setMinutes(int minutes) {
        getCalendarDate().setMinutes(minutes);
    }

    /**
     * Returns the number of seconds past the minute represented by this date.
     * The value returned is between <code>0</code> and <code>61</code>. The
     * values <code>60</code> and <code>61</code> can only occur on those
     * Java Virtual Machines that take leap seconds into account.
     *
     * @return  the number of seconds past the minute represented by this date.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.get(Calendar.SECOND)</code>.
     */
    @Deprecated
    public int getSeconds() {
        return normalize().getSeconds();
    }
```

```

    /**
     * Sets the seconds of this <tt>Date</tt> to the specified value.
     * This <tt>Date</tt> object is modified so that it represents a
     * point in time within the specified second of the minute, with
     * the year, month, date, hour, and minute the same as before, as
     * interpreted in the local time zone.
     *
     * @param   seconds   the seconds value.
     * @see     java.util.Calendar
     * @deprecated As of JDK version 1.1,
     * replaced by <code>Calendar.set(Calendar.SECOND, int seconds)</code>.
     */
    @Deprecated
    public void setSeconds(int seconds) {
        getCalendarDate().setSeconds(seconds);
    }

    /**
     * Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT
     * represented by this <tt>Date</tt> object.
     *
     * @return  the number of milliseconds since January 1, 1970, 00:00:00 GMT
     *          represented by this date.
     */
    public long getTime() {
        return getTimeImpl();
    }

    private final long getTimeImpl() {
        if (cdate != null && !cdate.isNormalized()) {
            normalize();
        }
        return fastTime;
    }

    /**
     * Sets this <code>Date</code> object to represent a point in time that is
     * <code>time</code> milliseconds after January 1, 1970 00:00:00 GMT.
     *
     * @param   time   the number of milliseconds.
     */
    public void setTime(long time) {
        fastTime = time;
        cdate = null;
    }

    /**
     * Tests if this date is before the specified date.
     *
     * @param   when   a date.
     * @return  <code>true</code> if and only if the instant of time
     *            represented by this <tt>Date</tt> object is strictly
     *            earlier than the instant represented by <tt>when</tt>;
     *          <code>false</code> otherwise.
```

```
     * @exception NullPointerException if <code>when</code> is null.
     */
    public boolean before(Date when) {
        return getMillisOf(this) < getMillisOf(when);
    }

    /**
     * Tests if this date is after the specified date.
     *
     * @param   when   a date.
     * @return  <code>true</code> if and only if the instant represented
     *          by this <tt>Date</tt> object is strictly later than the
     *          instant represented by <tt>when</tt>;
     *          <code>false</code> otherwise.
     * @exception NullPointerException if <code>when</code> is null.
     */
    public boolean after(Date when) {
        return getMillisOf(this) > getMillisOf(when);
    }

    /**
     * Compares two dates for equality.
     * The result is <code>true</code> if and only if the argument is
     * not <code>null</code> and is a <code>Date</code> object that
     * represents the same point in time, to the millisecond, as this object.
     * <p>
     * Thus, two <code>Date</code> objects are equal if and only if the
     * <code>getTime</code> method returns the same <code>long</code>
     * value for both.
     *
     * @param   obj   the object to compare with.
     * @return  <code>true</code> if the objects are the same;
     *          <code>false</code> otherwise.
     * @see     java.util.Date#getTime()
     */
    public boolean equals(Object obj) {
        return obj instanceof Date && getTime() == ((Date) obj).getTime();
    }

    /**
     * Returns the millisecond value of this <code>Date</code> object
     * without affecting its internal state.
     */
    static final long getMillisOf(Date date) {
        if (date.cdate == null || date.cdate.isNormalized()) {
            return date.fastTime;
        }
        BaseCalendar.Date d = (BaseCalendar.Date) date.cdate.clone();
        return gcal.getTime(d);
    }

    /**
     * Compares two Dates for ordering.
     *
     * @param   anotherDate   the <code>Date</code> to be compared.
```

```
     * @param   anotherDate   the <code>Date</code> to be compared.
     * @return  the value <code>0</code> if the argument Date is equal to
     *          this Date; a value less than <code>0</code> if this Date
     *          is before the Date argument; and a value greater than
     *      <code>0</code> if this Date is after the Date argument.
     * @since   1.2
     * @exception NullPointerException if <code>anotherDate</code> is null.
     */
    public int compareTo(Date anotherDate) {
        long thisTime = getMillisOf(this);
        long anotherTime = getMillisOf(anotherDate);
        return (thisTime<anotherTime ? -1 : (thisTime==anotherTime ? 0 : 1));
    }

    /**
     * Returns a hash code value for this object. The result is the
     * exclusive OR of the two halves of the primitive <tt>long</tt>
     * value returned by the {@link Date#getTime}
     * method. That is, the hash code is the value of the expression:
     * <blockquote><pre>
     * (int)(this.getTime()^(this.getTime() >>> 32))</pre></blockquote>
     *
     * @return  a hash code value for this object.
     */
    public int hashCode() {
        long ht = this.getTime();
        return (int) ht ^ (int) (ht >> 32);
    }

    /**
     * Converts this <code>Date</code> object to a <code>String</code>
     * of the form:
     * <blockquote><pre>
     * dow mon dd hh:mm:ss zzz yyyy</pre></blockquote>
     * where:<ul>
     * <li><tt>dow</tt> is the day of the week (<tt>Sun, Mon, Tue, Wed,
     *     Thu, Fri, Sat</tt>).
     * <li><tt>mon</tt> is the month (<tt>Jan, Feb, Mar, Apr, May, Jun,
     *     Jul, Aug, Sep, Oct, Nov, Dec</tt>).
     * <li><tt>dd</tt> is the day of the month (<tt>01</tt> through
     *     <tt>31</tt>), as two decimal digits.
     * <li><tt>hh</tt> is the hour of the day (<tt>00</tt> through
     *     <tt>23</tt>), as two decimal digits.
     * <li><tt>mm</tt> is the minute within the hour (<tt>00</tt> through
     *     <tt>59</tt>), as two decimal digits.
     * <li><tt>ss</tt> is the second within the minute (<tt>00</tt> through
     *     <tt>61</tt>, as two decimal digits.
     * <li><tt>zzz</tt> is the time zone (and may reflect daylight saving
     *     time). Standard time zone abbreviations include those
     *     recognized by the method <tt>parse</tt>. If time zone
     *     information is not available, then <tt>zzz</tt> is empty -
     *     that is, it consists of no characters at all.
     * <li><tt>yyyy</tt> is the year, as four decimal digits.
     * </ul>
     *
```

```
     * @return  a string representation of this date.
     * @see     java.util.Date#toLocaleString()
     * @see     java.util.Date#toGMTString()
     */
    public String toString() {
        // "EEE MMM dd HH:mm:ss zzz yyyy";
        BaseCalendar.Date date = normalize();
        StringBuilder sb = new StringBuilder(28);
        int index = date.getDayOfWeek();
        if (index == gcal.SUNDAY) {
            index = 8;
        }
        convertToAbbr(sb, wtb[index]).append(' ');                        // EEE
        convertToAbbr(sb, wtb[date.getMonth() - 1 + 2 + 7]).append(' ');  // MMM
        CalendarUtils.sprintf0d(sb, date.getDayOfMonth(), 2).append(' '); // dd

        CalendarUtils.sprintf0d(sb, date.getHours(), 2).append(':');   // HH
        CalendarUtils.sprintf0d(sb, date.getMinutes(), 2).append(':'); // mm
        CalendarUtils.sprintf0d(sb, date.getSeconds(), 2).append(' '); // ss
        TimeZone zi = date.getZone();
        if (zi != null) {
            sb.append(zi.getDisplayName(date.isDaylightTime(), zi.SHORT, Locale.US)); // zzz
        } else {
            sb.append("GMT");
        }
        sb.append(' ').append(date.getYear());  // yyyy
        return sb.toString();
    }

    /**
     * Converts the given name to its 3-letter abbreviation (e.g.,
     * "monday" -> "Mon") and stored the abbreviation in the given
     * <code>StringBuilder</code>.
     */
    private static final StringBuilder convertToAbbr(StringBuilder sb, String name) {
        sb.append(Character.toUpperCase(name.charAt(0)));
        sb.append(name.charAt(1)).append(name.charAt(2));
        return sb;
    }

    /**
     * Creates a string representation of this <tt>Date</tt> object in an
     * implementation-dependent form. The intent is that the form should
     * be familiar to the user of the Java application, wherever it may
     * happen to be running. The intent is comparable to that of the
     * "<code>%c</code>" format supported by the <code>strftime()</code>
     * function of ISO C.
     *
     * @return  a string representation of this date, using the locale
     *          conventions.
     * @see     java.text.DateFormat
     * @see     java.util.Date#toString()
     * @see     java.util.Date#toGMTString()
     * @deprecated As of JDK version 1.1,
     * replaced by <code>DateFormat.format(Date date)</code>.
```

```
     * replaced by <code>DateFormat.format(Date date)</code>.
     */
    @Deprecated
    public String toLocaleString() {
        DateFormat formatter = DateFormat.getDateTimeInstance();
        return formatter.format(this);
    }

    /**
     * Creates a string representation of this <tt>Date</tt> object of
     * the form:
     * <blockquote<pre>
     * d mon yyyy hh:mm:ss GMT</pre></blockquote>
     * where:<ul>
     * <li><i>d</i> is the day of the month (<tt>1</tt> through <tt>31</tt>),
     *     as one or two decimal digits.
     * <li><i>mon</i> is the month (<tt>Jan, Feb, Mar, Apr, May, Jun, Jul,
     *     Aug, Sep, Oct, Nov, Dec</tt>).
     * <li><i>yyyy</i> is the year, as four decimal digits.
     * <li><i>hh</i> is the hour of the day (<tt>00</tt> through <tt>23</tt>),
     *     as two decimal digits.
     * <li><i>mm</i> is the minute within the hour (<tt>00</tt> through
     *     <tt>59</tt>), as two decimal digits.
     * <li><i>ss</i> is the second within the minute (<tt>00</tt> through
     *     <tt>61</tt>), as two decimal digits.
     * <li><i>GMT</i> is exactly the ASCII letters "<tt>GMT</tt>" to indicate
     *     Greenwich Mean Time.
     * </ul><p>
     * The result does not depend on the local time zone.
     *
     * @return  a string representation of this date, using the Internet GMT
     *          conventions.
     * @see     java.text.DateFormat
     * @see     java.util.Date#toString()
     * @see     java.util.Date#toLocaleString()
     * @deprecated As of JDK version 1.1,
     * replaced by <code>DateFormat.format(Date date)</code>, using a
     * GMT <code>TimeZone</code>.
     */
    @Deprecated
    public String toGMTString() {
        // d MMM yyyy HH:mm:ss 'GMT'
        long t = getTime();
        BaseCalendar cal = getCalendarSystem(t);
        BaseCalendar.Date date =
            (BaseCalendar.Date) cal.getCalendarDate(getTime(), (TimeZone)null);
        StringBuilder sb = new StringBuilder(32);
        CalendarUtils.sprintf0d(sb, date.getDayOfMonth(), 1).append(' '); // d
        convertToAbbr(sb, wtb[date.getMonth() - 1 + 2 + 7]).append(' ');  // MMM
        sb.append(date.getYear()).append(' ');                            // yyyy
        CalendarUtils.sprintf0d(sb, date.getHours(), 2).append(':');      // HH
        CalendarUtils.sprintf0d(sb, date.getMinutes(), 2).append(':');    // mm
        CalendarUtils.sprintf0d(sb, date.getSeconds(), 2);                // ss
        sb.append(" GMT");                                                // ' GMT'
        return sb.toString();
```

```
    }

    /**
     * Returns the offset, measured in minutes, for the local time zone
     * relative to UTC that is appropriate for the time represented by
     * this <code>Date</code> object.
     * <p>
     * For example, in Massachusetts, five time zones west of Greenwich:
     * <blockquote><pre>
     * new Date(96, 1, 14).getTimezoneOffset() returns 300</pre></blockquote>
     * because on February 14, 1996, standard time (Eastern Standard Time)
     * is in use, which is offset five hours from UTC; but:
     * <blockquote><pre>
     * new Date(96, 5, 1).getTimezoneOffset() returns 240</pre></blockquote>
     * because on June 1, 1996, daylight saving time (Eastern Daylight Time)
     * is in use, which is offset only four hours from UTC.<p>
     * This method produces the same result as if it computed:
     * <blockquote><pre>
     * (this.getTime() - UTC(this.getYear(),
     *                       this.getMonth(),
     *                       this.getDate(),
     *                       this.getHours(),
     *                       this.getMinutes(),
     *                       this.getSeconds())) / (60 * 1000)
     * </pre></blockquote>
     *
     * @return  the time-zone offset, in minutes, for the current time zone.
     * @see     java.util.Calendar#ZONE_OFFSET
     * @see     java.util.Calendar#DST_OFFSET
     * @see     java.util.TimeZone#getDefault
     * @deprecated As of JDK version 1.1,
     * replaced by <code>-(Calendar.get(Calendar.ZONE_OFFSET) +
     * Calendar.get(Calendar.DST_OFFSET)) / (60 * 1000)</code>.
     */
    @Deprecated
    public int getTimezoneOffset() {
        int zoneOffset;
        if (cdate == null) {
            TimeZone tz = TimeZone.getDefaultRef();
            if (tz instanceof ZoneInfo) {
                zoneOffset = ((ZoneInfo)tz).getOffsets(fastTime, null);
            } else {
                zoneOffset = tz.getOffset(fastTime);
            }
        } else {
            normalize();
            zoneOffset = cdate.getZoneOffset();
        }
        return -zoneOffset/60000;  // convert to minutes
    }

    private final BaseCalendar.Date getCalendarDate() {
        if (cdate == null) {
            BaseCalendar cal = getCalendarSystem(fastTime);
            cdate = (BaseCalendar.Date) cal.getCalendarDate(fastTime,
```

```
                                                                TimeZone.getDefaultRef());
            }
            return cdate;
        }

        private final BaseCalendar.Date normalize() {
            if (cdate == null) {
                BaseCalendar cal = getCalendarSystem(fastTime);
                cdate = (BaseCalendar.Date) cal.getCalendarDate(fastTime,
                                                                TimeZone.getDefaultRef());
                return cdate;
            }

            // Normalize cdate with the TimeZone in cdate first. This is
            // required for the compatible behavior.
            if (!cdate.isNormalized()) {
                cdate = normalize(cdate);
            }

            // If the default TimeZone has changed, then recalculate the
            // fields with the new TimeZone.
            TimeZone tz = TimeZone.getDefaultRef();
            if (tz != cdate.getZone()) {
                cdate.setZone(tz);
                CalendarSystem cal = getCalendarSystem(cdate);
                cal.getCalendarDate(fastTime, cdate);
            }
            return cdate;
        }

        // fastTime and the returned data are in sync upon return.
        private final BaseCalendar.Date normalize(BaseCalendar.Date date) {
            int y = date.getNormalizedYear();
            int m = date.getMonth();
            int d = date.getDayOfMonth();
            int hh = date.getHours();
            int mm = date.getMinutes();
            int ss = date.getSeconds();
            int ms = date.getMillis();
            TimeZone tz = date.getZone();

            // If the specified year can't be handled using a long value
            // in milliseconds, GregorianCalendar is used for full
            // compatibility with underflow and overflow. This is required
            // by some JCK tests. The limits are based max year values -
            // years that can be represented by max values of d, hh, mm,
            // ss and ms. Also, let GregorianCalendar handle the default
            // cutover year so that we don't need to worry about the
            // transition here.
            if (y == 1582 || y > 280000000 || y < -280000000) {
                if (tz == null) {
                    tz = TimeZone.getTimeZone("GMT");
                }
                GregorianCalendar gc = new GregorianCalendar(tz);
                gc.clear();
```

```
                gc.clear();
                gc.set(gc.MILLISECOND, ms);
                gc.set(y, m-1, d, hh, mm, ss);
                fastTime = gc.getTimeInMillis();
                BaseCalendar cal = getCalendarSystem(fastTime);
                date = (BaseCalendar.Date) cal.getCalendarDate(fastTime, tz);
                return date;
            }

            BaseCalendar cal = getCalendarSystem(y);
            if (cal != getCalendarSystem(date)) {
                date = (BaseCalendar.Date) cal.newCalendarDate(tz);
                date.setNormalizedDate(y, m, d).setTimeOfDay(hh, mm, ss, ms);
            }
            // Perform the GregorianCalendar-style normalization.
            fastTime = cal.getTime(date);

            // In case the normalized date requires the other calendar
            // system, we need to recalculate it using the other one.
            BaseCalendar ncal = getCalendarSystem(fastTime);
            if (ncal != cal) {
                date = (BaseCalendar.Date) ncal.newCalendarDate(tz);
                date.setNormalizedDate(y, m, d).setTimeOfDay(hh, mm, ss, ms);
                fastTime = ncal.getTime(date);
            }
            return date;
        }

        /**
         * Returns the Gregorian or Julian calendar system to use with the
         * given date. Use Gregorian from October 15, 1582.
         *
         * @param year normalized calendar year (not -1900)
         * @return the CalendarSystem to use for the specified date
         */
        private static final BaseCalendar getCalendarSystem(int year) {
            if (year >= 1582) {
                return gcal;
            }
            return getJulianCalendar();
        }

        private static final BaseCalendar getCalendarSystem(long utc) {
            // Quickly check if the time stamp given by `utc' is the Epoch
            // or later. If it's before 1970, we convert the cutover to
            // local time to compare.
            if (utc >= 0
                || utc >= GregorianCalendar.DEFAULT_GREGORIAN_CUTOVER
                            - TimeZone.getDefaultRef().getOffset(utc)) {
                return gcal;
            }
            return getJulianCalendar();
        }

        private static final BaseCalendar getCalendarSystem(BaseCalendar.Date cdate) {
```

```
            if (jcal == null) {
                return gcal;
            }
            if (cdate.getEra() != null) {
                return jcal;
            }
            return gcal;
        }

        synchronized private static final BaseCalendar getJulianCalendar() {
            if (jcal == null) {
                jcal = (BaseCalendar) CalendarSystem.forName("julian");
            }
            return jcal;
        }

        /**
         * Save the state of this object to a stream (i.e., serialize it).
         *
         * @serialData The value returned by <code>getTime()</code>
         *             is emitted (long).  This represents the offset from
         *             January 1, 1970, 00:00:00 GMT in milliseconds.
         */
        private void writeObject(ObjectOutputStream s)
             throws IOException
        {
            s.writeLong(getTimeImpl());
        }

        /**
         * Reconstitute this object from a stream (i.e., deserialize it).
         */
        private void readObject(ObjectInputStream s)
             throws IOException, ClassNotFoundException
        {
            fastTime = s.readLong();
        }
    }
```