Appendix B
`org.jfree.date.SerialDate`

Listing B-1

`SerialDate.Java`

```
/* ===========================================================
 * JCommon : a free general purpose class library for the Java(tm) platform
 * ===========================================================
 *
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 *
 * Project Info:  http://www.jfree.org/jcommon/index.html
 *
 * This library is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This library 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 Lesser General Public
 * License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 * USA.
 *
 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
 * in the United States and other countries.]
 *
 * --------------
 * SerialDate.java
 * --------------
 * (C) Copyright 2001-2005, by Object Refinery Limited.
 *
 * Original Author:  David Gilbert (for Object Refinery Limited);
 * Contributor(s):   -;
 *
 * $Id: SerialDate.java,v 1.7 2005/11/03 09:25:17 mungady Exp $
 *
 * Changes (from 11-Oct-2001)
```

```
 * --------------------------
 * 11-Oct-2001 : Re-organised the class and moved it to new package
 *               com.jrefinery.date (DG);
 * 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
 *               class (DG);
 * 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
 *               class is gone (DG);  Changed getPreviousDayOfWeek(),
 *               getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
 *               bugs (DG);
 * 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
 * 29-May-2002 : Moved the month constants into a separate interface
 *               (MonthConstants) (DG);
 * 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
 * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 * 13-Mar-2003 : Implemented Serializable (DG);
 * 29-May-2003 : Fixed bug in addMonths method (DG);
 * 04-Sep-2003 : Implemented Comparable.  Updated the isInRange javadocs (DG);
 * 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
 *
 */

package org.jfree.date;

import java.io.Serializable;
import java.text.DateFormatSymbols;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.GregorianCalendar;

/**
 *  An abstract class that defines our requirements for manipulating dates,
 *  without tying down a particular implementation.
 *  <P>
 *  Requirement 1 : match at least what Excel does for dates;
 *  Requirement 2 : class is immutable;
 *  <P>
 *  Why not just use java.util.Date?  We will, when it makes sense.  At times,
 *  java.util.Date can be *too* precise - it represents an instant in time,
 *  accurate to 1/1000th of a second (with the date itself depending on the
 *  time-zone).  Sometimes we just want to represent a particular day (e.g. 21
 *  January 2015) without concerning ourselves about the time of day, or the
 *  time-zone, or anything else.  That's what we've defined SerialDate for.
 *  <P>
 *  You can call getInstance() to get a concrete subclass of SerialDate,
 *  without worrying about the exact implementation.
 *
 * @author David Gilbert
 */
public abstract class SerialDate implements Comparable,
                                            Serializable,
                                            MonthConstants {

    /** For serialization. */
    private static final long serialVersionUID = -293716040467423637L;

    /** Date format symbols. */
    public static final DateFormatSymbols
        DATE_FORMAT_SYMBOLS = new SimpleDateFormat().getDateFormatSymbols();

    /** The serial number for 1 January 1900. */
    public static final int SERIAL_LOWER_BOUND = 2;

```

```
100     /** The serial number for 31 December 9999. */
101     public static final int SERIAL_UPPER_BOUND = 2958465;
102
103     /** The lowest year value supported by this date format. */
104     public static final int MINIMUM_YEAR_SUPPORTED = 1900;
105
106     /** The highest year value supported by this date format. */
107     public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
108
109     /** Useful constant for Monday. Equivalent to java.util.Calendar.MONDAY. */
110     public static final int MONDAY = Calendar.MONDAY;
111
112     /**
113      * Useful constant for Tuesday. Equivalent to java.util.Calendar.TUESDAY.
114      */
115     public static final int TUESDAY = Calendar.TUESDAY;
116
117     /**
118      * Useful constant for Wednesday. Equivalent to
119      * java.util.Calendar.WEDNESDAY.
120      */
121     public static final int WEDNESDAY = Calendar.WEDNESDAY;
122
123     /**
124      * Useful constant for Thrusday. Equivalent to java.util.Calendar.THURSDAY.
125      */
126     public static final int THURSDAY = Calendar.THURSDAY;
127
128     /** Useful constant for Friday. Equivalent to java.util.Calendar.FRIDAY. */
129     public static final int FRIDAY = Calendar.FRIDAY;
130
131     /**
132      * Useful constant for Saturday. Equivalent to java.util.Calendar.SATURDAY.
133      */
134     public static final int SATURDAY = Calendar.SATURDAY;
135
136     /** Useful constant for Sunday. Equivalent to java.util.Calendar.SUNDAY. */
137     public static final int SUNDAY = Calendar.SUNDAY;
138
139     /** The number of days in each month in non leap years. */
140     static final int[] LAST_DAY_OF_MONTH =
141         {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
142
143     /** The number of days in a (non-leap) year up to the end of each month. */
144     static final int[] AGGREGATE_DAYS_TO_END_OF_MONTH =
145         {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
146
147     /** The number of days in a year up to the end of the preceding month. */
148     static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
149         {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
150
151     /** The number of days in a leap year up to the end of each month. */
152     static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH =
153         {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
154
155     /**
156      * The number of days in a leap year up to the end of the preceding month.
157      */
158     static final int[]
159         LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
160             {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
161
```

```
162     /** A useful constant for referring to the first week in a month. */
163     public static final int FIRST_WEEK_IN_MONTH = 1;
164
165     /** A useful constant for referring to the second week in a month. */
166     public static final int SECOND_WEEK_IN_MONTH = 2;
167
168     /** A useful constant for referring to the third week in a month. */
169     public static final int THIRD_WEEK_IN_MONTH = 3;
170
171     /** A useful constant for referring to the fourth week in a month. */
172     public static final int FOURTH_WEEK_IN_MONTH = 4;
173
174     /** A useful constant for referring to the last week in a month. */
175     public static final int LAST_WEEK_IN_MONTH = 0;
176
177     /** Useful range constant. */
178     public static final int INCLUDE_NONE = 0;
179
180     /** Useful range constant. */
181     public static final int INCLUDE_FIRST = 1;
182
183     /** Useful range constant. */
184     public static final int INCLUDE_SECOND = 2;
185
186     /** Useful range constant. */
187     public static final int INCLUDE_BOTH = 3;
188
189     /**
190      * Useful constant for specifying a day of the week relative to a fixed
191      * date.
192      */
193     public static final int PRECEDING = -1;
194
195     /**
196      * Useful constant for specifying a day of the week relative to a fixed
197      * date.
198      */
199     public static final int NEAREST = 0;
200
201     /**
202      * Useful constant for specifying a day of the week relative to a fixed
203      * date.
204      */
205     public static final int FOLLOWING = 1;
206
207     /** A description for the date. */
208     private String description;
209
210     /**
211      * Default constructor.
212      */
213     protected SerialDate() {
214     }
215
216     /**
217      * Returns <code>true</code> if the supplied integer code represents a
218      * valid day-of-the-week, and <code>false</code> otherwise.
219      *
220      * @param code  the code being checked for validity.
221      *
222      * @return <code>true</code> if the supplied integer code represents a
223      *         valid day-of-the-week, and <code>false</code> otherwise.
```

```
224      */
225     public static boolean isValidWeekdayCode(final int code) {
226
227         switch(code) {
228             case SUNDAY:
229             case MONDAY:
230             case TUESDAY:
231             case WEDNESDAY:
232             case THURSDAY:
233             case FRIDAY:
234             case SATURDAY:
235                 return true;
236             default:
237                 return false;
238         }
239
240     }
241
242     /**
243      * Converts the supplied string to a day of the week.
244      *
245      * @param s  a string representing the day of the week.
246      *
247      * @return <code>-1</code> if the string is not convertable, the day of
248      *         the week otherwise.
249      */
250     public static int stringToWeekdayCode(String s) {
251
252         final String[] shortWeekdayNames
253             = DATE_FORMAT_SYMBOLS.getShortWeekdays();
254         final String[] weekDayNames = DATE_FORMAT_SYMBOLS.getWeekdays();
255
256         int result = -1;
257         s = s.trim();
258         for (int i = 0; i < weekDayNames.length; i++) {
259             if (s.equals(shortWeekdayNames[i])) {
260                 result = i;
261                 break;
262             }
263             if (s.equals(weekDayNames[i])) {
264                 result = i;
265                 break;
266             }
267         }
268         return result;
269
270     }
271
272     /**
273      * Returns a string representing the supplied day-of-the-week.
274      * <P>
275      * Need to find a better approach.
276      *
277      * @param weekday  the day of the week.
278      *
279      * @return a string representing the supplied day-of-the-week.
280      */
281     public static String weekdayCodeToString(final int weekday) {
282
283         final String[] weekdays = DATE_FORMAT_SYMBOLS.getWeekdays();
284         return weekdays[weekday];
285
```

```
286     }
287
288     /**
289      * Returns an array of month names.
290      *
291      * @return an array of month names.
292      */
293     public static String[] getMonths() {
294
295         return getMonths(false);
296
297     }
298
299     /**
300      * Returns an array of month names.
301      *
302      * @param shortened  a flag indicating that shortened month names should
303      *                   be returned.
304      *
305      * @return an array of month names.
306      */
307     public static String[] getMonths(final boolean shortened) {
308
309         if (shortened) {
310             return DATE_FORMAT_SYMBOLS.getShortMonths();
311         }
312         else {
313             return DATE_FORMAT_SYMBOLS.getMonths();
314         }
315
316     }
317
318     /**
319      * Returns true if the supplied integer code represents a valid month.
320      *
321      * @param code  the code being checked for validity.
322      *
323      * @return <code>true</code> if the supplied integer code represents a
324      *         valid month.
325      */
326     public static boolean isValidMonthCode(final int code) {
327
328         switch(code) {
329             case JANUARY:
330             case FEBRUARY:
331             case MARCH:
332             case APRIL:
333             case MAY:
334             case JUNE:
335             case JULY:
336             case AUGUST:
337             case SEPTEMBER:
338             case OCTOBER:
339             case NOVEMBER:
340             case DECEMBER:
341                 return true;
342             default:
343                 return false;
344         }
345
346     }
347
```

```
348     /**
349      * Returns the quarter for the specified month.
350      *
351      * @param code  the month code (1-12).
352      *
353      * @return the quarter that the month belongs to.
354      * @throws java.lang.IllegalArgumentException
355      */
356     public static int monthCodeToQuarter(final int code) {
357
358         switch(code) {
359             case JANUARY:
360             case FEBRUARY:
361             case MARCH: return 1;
362             case APRIL:
363             case MAY:
364             case JUNE: return 2;
365             case JULY:
366             case AUGUST:
367             case SEPTEMBER: return 3;
368             case OCTOBER:
369             case NOVEMBER:
370             case DECEMBER: return 4;
371             default: throw new IllegalArgumentException(
372                 "SerialDate.monthCodeToQuarter: invalid month code.");
373         }
374
375     }
376
377     /**
378      * Returns a string representing the supplied month.
379      * <P>
380      * The string returned is the long form of the month name taken from the
381      * default locale.
382      *
383      * @param month  the month.
384      *
385      * @return a string representing the supplied month.
386      */
387     public static String monthCodeToString(final int month) {
388
389         return monthCodeToString(month, false);
390
391     }
392
393     /**
394      * Returns a string representing the supplied month.
395      * <P>
396      * The string returned is the long or short form of the month name taken
397      * from the default locale.
398      *
399      * @param month  the month.
400      * @param shortened  if <code>true</code> return the abbreviation of the
401      *                   month.
402      *
403      * @return a string representing the supplied month.
404      * @throws java.lang.IllegalArgumentException
405      */
406     public static String monthCodeToString(final int month,
407                                            final boolean shortened) {
408
409         // check arguments...
```

```
410         if (!isValidMonthCode(month)) {
411             throw new IllegalArgumentException(
412                 "SerialDate.monthCodeToString: month outside valid range.");
413         }
414
415         final String[] months;
416
417         if (shortened) {
418             months = DATE_FORMAT_SYMBOLS.getShortMonths();
419         }
420         else {
421             months = DATE_FORMAT_SYMBOLS.getMonths();
422         }
423
424         return months[month - 1];
425
426     }
427
428     /**
429      * Converts a string to a month code.
430      * <P>
431      * This method will return one of the constants JANUARY, FEBRUARY, ...,
432      * DECEMBER that corresponds to the string.  If the string is not
433      * recognised, this method returns -1.
434      *
435      * @param s  the string to parse.
436      *
437      * @return <code>-1</code> if the string is not parseable, the month of the
438      *         year otherwise.
439      */
440     public static int stringToMonthCode(String s) {
441
442         final String[] shortMonthNames = DATE_FORMAT_SYMBOLS.getShortMonths();
443         final String[] monthNames = DATE_FORMAT_SYMBOLS.getMonths();
444
445         int result = -1;
446         s = s.trim();
447
448         // first try parsing the string as an integer (1-12)...
449         try {
450             result = Integer.parseInt(s);
451         }
452         catch (NumberFormatException e) {
453             // suppress
454         }
455
456         // now search through the month names...
457         if ((result < 1) || (result > 12)) {
458             for (int i = 0; i < monthNames.length; i++) {
459                 if (s.equals(shortMonthNames[i])) {
460                     result = i + 1;
461                     break;
462                 }
463                 if (s.equals(monthNames[i])) {
464                     result = i + 1;
465                     break;
466                 }
467             }
468         }
469
470         return result;
471
```

```
472     }
473
474     /**
475      * Returns true if the supplied integer code represents a valid
476      * week-in-the-month, and false otherwise.
477      *
478      * @param code  the code being checked for validity.
479      * @return <code>true</code> if the supplied integer code represents a
480      *         valid week-in-the-month.
481      */
482     public static boolean isValidWeekInMonthCode(final int code) {
483
484         switch(code) {
485             case FIRST_WEEK_IN_MONTH:
486             case SECOND_WEEK_IN_MONTH:
487             case THIRD_WEEK_IN_MONTH:
488             case FOURTH_WEEK_IN_MONTH:
489             case LAST_WEEK_IN_MONTH: return true;
490             default: return false;
491         }
492
493     }
494
495     /**
496      * Determines whether or not the specified year is a leap year.
497      *
498      * @param yyyy  the year (in the range 1900 to 9999).
499      *
500      * @return <code>true</code> if the specified year is a leap year.
501      */
502     public static boolean isLeapYear(final int yyyy) {
503
504         if ((yyyy % 4) != 0) {
505             return false;
506         }
507         else if ((yyyy % 400) == 0) {
508             return true;
509         }
510         else if ((yyyy % 100) == 0) {
511             return false;
512         }
513         else {
514             return true;
515         }
516
517     }
518
519     /**
520      * Returns the number of leap years from 1900 to the specified year
521      * INCLUSIVE.
522      * <P>
523      * Note that 1900 is not a leap year.
524      *
525      * @param yyyy  the year (in the range 1900 to 9999).
526      *
527      * @return the number of leap years from 1900 to the specified year.
528      */
529     public static int leapYearCount(final int yyyy) {
530
531         final int leap4 = (yyyy - 1896) / 4;
532         final int leap100 = (yyyy - 1800) / 100;
533         final int leap400 = (yyyy - 1600) / 400;
```

```
534         return leap4 - leap100 + leap400;
535
536     }
537
538     /**
539      * Returns the number of the last day of the month, taking into account
540      * leap years.
541      *
542      * @param month  the month.
543      * @param yyyy  the year (in the range 1900 to 9999).
544      *
545      * @return the number of the last day of the month.
546      */
547     public static int lastDayOfMonth(final int month, final int yyyy) {
548
549         final int result = LAST_DAY_OF_MONTH[month];
550         if (month != FEBRUARY) {
551             return result;
552         }
553         else if (isLeapYear(yyyy)) {
554             return result + 1;
555         }
556         else {
557             return result;
558         }
559
560     }
561
562     /**
563      * Creates a new date by adding the specified number of days to the base
564      * date.
565      *
566      * @param days  the number of days to add (can be negative).
567      * @param base  the base date.
568      *
569      * @return a new date.
570      */
571     public static SerialDate addDays(final int days, final SerialDate base) {
572
573         final int serialDayNumber = base.toSerial() + days;
574         return SerialDate.createInstance(serialDayNumber);
575
576     }
577
578     /**
579      * Creates a new date by adding the specified number of months to the base
580      * date.
581      * <P>
582      * If the base date is close to the end of the month, the day on the result
583      * may be adjusted slightly:  31 May + 1 month = 30 June.
584      *
585      * @param months  the number of months to add (can be negative).
586      * @param base  the base date.
587      *
588      * @return a new date.
589      */
590     public static SerialDate addMonths(final int months,
591                                        final SerialDate base) {
592
593         final int yy = (12 * base.getYYYY() + base.getMonth() + months - 1)
594                        / 12;
595         final int mm = (12 * base.getYYYY() + base.getMonth() + months - 1)
```

```
596                         % 12 + 1;
597         final int dd = Math.min(
598             base.getDayOfMonth(), SerialDate.lastDayOfMonth(mm, yy)
599         );
600         return SerialDate.createInstance(dd, mm, yy);
601
602     }
603
604     /**
605      * Creates a new date by adding the specified number of years to the base
606      * date.
607      *
608      * @param years  the number of years to add (can be negative).
609      * @param base  the base date.
610      *
611      * @return A new date.
612      */
613     public static SerialDate addYears(final int years, final SerialDate base) {
614
615         final int baseY = base.getYYYY();
616         final int baseM = base.getMonth();
617         final int baseD = base.getDayOfMonth();
618
619         final int targetY = baseY + years;
620         final int targetD = Math.min(
621             baseD, SerialDate.lastDayOfMonth(baseM, targetY)
622         );
623
624         return SerialDate.createInstance(targetD, baseM, targetY);
625
626     }
627
628     /**
629      * Returns the latest date that falls on the specified day-of-the-week and
630      * is BEFORE the base date.
631      *
632      * @param targetWeekday  a code for the target day-of-the-week.
633      * @param base  the base date.
634      *
635      * @return the latest date that falls on the specified day-of-the-week and
636      *         is BEFORE the base date.
637      */
638     public static SerialDate getPreviousDayOfWeek(final int targetWeekday,
639                                                    final SerialDate base) {
640
641         // check arguments...
642         if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
643             throw new IllegalArgumentException(
644                 "Invalid day-of-the-week code."
645             );
646         }
647
648         // find the date...
649         final int adjust;
650         final int baseDOW = base.getDayOfWeek();
651         if (baseDOW > targetWeekday) {
652             adjust = Math.min(0, targetWeekday - baseDOW);
653         }
654         else {
655             adjust = -7 + Math.max(0, targetWeekday - baseDOW);
656         }
657
```

```
658         return SerialDate.addDays(adjust, base);
659
660     }
661
662     /**
663      * Returns the earliest date that falls on the specified day-of-the-week
664      * and is AFTER the base date.
665      *
666      * @param targetWeekday  a code for the target day-of-the-week.
667      * @param base  the base date.
668      *
669      * @return the earliest date that falls on the specified day-of-the-week
670      *         and is AFTER the base date.
671      */
672     public static SerialDate getFollowingDayOfWeek(final int targetWeekday,
673                                                     final SerialDate base) {
674
675         // check arguments...
676         if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
677             throw new IllegalArgumentException(
678                 "Invalid day-of-the-week code."
679             );
680         }
681
682         // find the date...
683         final int adjust;
684         final int baseDOW = base.getDayOfWeek();
685         if (baseDOW > targetWeekday) {
686             adjust = 7 + Math.min(0, targetWeekday - baseDOW);
687         }
688         else {
689             adjust = Math.max(0, targetWeekday - baseDOW);
690         }
691
692         return SerialDate.addDays(adjust, base);
693     }
694
695     /**
696      * Returns the date that falls on the specified day-of-the-week and is
697      * CLOSEST to the base date.
698      *
699      * @param targetDOW  a code for the target day-of-the-week.
700      * @param base  the base date.
701      *
702      * @return the date that falls on the specified day-of-the-week and is
703      *         CLOSEST to the base date.
704      */
705     public static SerialDate getNearestDayOfWeek(final int targetDOW,
706                                                   final SerialDate base) {
707
708         // check arguments...
709         if (!SerialDate.isValidWeekdayCode(targetDOW)) {
710             throw new IllegalArgumentException(
711                 "Invalid day-of-the-week code."
712             );
713         }
714
715         // find the date...
716         final int baseDOW = base.getDayOfWeek();
717         int adjust = -Math.abs(targetDOW - baseDOW);
718         if (adjust >= 4) {
719             adjust = 7 - adjust;
```

```
720         }
721         if (adjust <= -4) {
722             adjust = 7 + adjust;
723         }
724         return SerialDate.addDays(adjust, base);
725
726     }
727
728     /**
729      * Rolls the date forward to the last day of the month.
730      *
731      * @param base  the base date.
732      *
733      * @return a new serial date.
734      */
735     public SerialDate getEndOfCurrentMonth(final SerialDate base) {
736         final int last = SerialDate.lastDayOfMonth(
737             base.getMonth(), base.getYYYY()
738         );
739         return SerialDate.createInstance(last, base.getMonth(), base.getYYYY());
740     }
741
742     /**
743      * Returns a string corresponding to the week-in-the-month code.
744      * <P>
745      * Need to find a better approach.
746      *
747      * @param count  an integer code representing the week-in-the-month.
748      *
749      * @return a string corresponding to the week-in-the-month code.
750      */
751     public static String weekInMonthToString(final int count) {
752
753         switch (count) {
754             case SerialDate.FIRST_WEEK_IN_MONTH : return "First";
755             case SerialDate.SECOND_WEEK_IN_MONTH : return "Second";
756             case SerialDate.THIRD_WEEK_IN_MONTH : return "Third";
757             case SerialDate.FOURTH_WEEK_IN_MONTH : return "Fourth";
758             case SerialDate.LAST_WEEK_IN_MONTH : return "Last";
759             default :
760                 return "SerialDate.weekInMonthToString(): invalid code.";
761         }
762
763     }
764
765     /**
766      * Returns a string representing the supplied 'relative'.
767      * <P>
768      * Need to find a better approach.
769      *
770      * @param relative  a constant representing the 'relative'.
771      *
772      * @return a string representing the supplied 'relative'.
773      */
774     public static String relativeToString(final int relative) {
775
776         switch (relative) {
777             case SerialDate.PRECEDING : return "Preceding";
778             case SerialDate.NEAREST : return "Nearest";
779             case SerialDate.FOLLOWING : return "Following";
780             default : return "ERROR : Relative To String";
781         }
```

```
782
783     }
784
785     /**
786      * Factory method that returns an instance of some concrete subclass of
787      * {@link SerialDate}.
788      *
789      * @param day  the day (1-31).
790      * @param month  the month (1-12).
791      * @param yyyy  the year (in the range 1900 to 9999).
792      *
793      * @return An instance of {@link SerialDate}.
794      */
795     public static SerialDate createInstance(final int day, final int month,
796                                             final int yyyy) {
797         return new SpreadsheetDate(day, month, yyyy);
798     }
799
800     /**
801      * Factory method that returns an instance of some concrete subclass of
802      * {@link SerialDate}.
803      *
804      * @param serial  the serial number for the day (1 January 1900 = 2).
805      *
806      * @return a instance of SerialDate.
807      */
808     public static SerialDate createInstance(final int serial) {
809         return new SpreadsheetDate(serial);
810     }
811
812     /**
813      * Factory method that returns an instance of a subclass of SerialDate.
814      *
815      * @param date  A Java date object.
816      *
817      * @return a instance of SerialDate.
818      */
819     public static SerialDate createInstance(final java.util.Date date) {
820
821         final GregorianCalendar calendar = new GregorianCalendar();
822         calendar.setTime(date);
823         return new SpreadsheetDate(calendar.get(Calendar.DATE),
824                                    calendar.get(Calendar.MONTH) + 1,
825                                    calendar.get(Calendar.YEAR));
826
827     }
828
829     /**
830      * Returns the serial number for the date, where 1 January 1900 = 2 (this
831      * corresponds, almost, to the numbering system used in Microsoft Excel for
832      * Windows and Lotus 1-2-3).
833      *
834      * @return the serial number for the date.
835      */
836     public abstract int toSerial();
837
838     /**
839      * Returns a java.util.Date.  Since java.util.Date has more precision than
840      * SerialDate, we need to define a convention for the 'time of day'.
841      *
842      * @return this as <code>java.util.Date</code>.
843      */
```

```
844     public abstract java.util.Date toDate();
845
846     /**
847      * Returns a description of the date.
848      *
849      * @return a description of the date.
850      */
851     public String getDescription() {
852         return this.description;
853     }
854
855     /**
856      * Sets the description for the date.
857      *
858      * @param description  the new description for the date.
859      */
860     public void setDescription(final String description) {
861         this.description = description;
862     }
863
864     /**
865      * Converts the date to a string.
866      *
867      * @return  a string representation of the date.
868      */
869     public String toString() {
870         return getDayOfMonth() + "-" + SerialDate.monthCodeToString(getMonth())
871                                + "-" + getYYYY();
872     }
873
874     /**
875      * Returns the year (assume a valid range of 1900 to 9999).
876      *
877      * @return the year.
878      */
879     public abstract int getYYYY();
880
881     /**
882      * Returns the month (January = 1, February = 2, March = 3).
883      *
884      * @return the month of the year.
885      */
886     public abstract int getMonth();
887
888     /**
889      * Returns the day of the month.
890      *
891      * @return the day of the month.
892      */
893     public abstract int getDayOfMonth();
894
895     /**
896      * Returns the day of the week.
897      *
898      * @return the day of the week.
899      */
900     public abstract int getDayOfWeek();
901
902     /**
903      * Returns the difference (in days) between this date and the specified
904      * 'other' date.
905      * <P>
```

```
906      * The result is positive if this date is after the 'other' date and
907      * negative if it is before the 'other' date.
908      *
909      * @param other  the date being compared to.
910      *
911      * @return the difference between this and the other date.
912      */
913     public abstract int compare(SerialDate other);
914
915     /**
916      * Returns true if this SerialDate represents the same date as the
917      * specified SerialDate.
918      *
919      * @param other  the date being compared to.
920      *
921      * @return <code>true</code> if this SerialDate represents the same date as
922      *         the specified SerialDate.
923      */
924     public abstract boolean isOn(SerialDate other);
925
926     /**
927      * Returns true if this SerialDate represents an earlier date compared to
928      * the specified SerialDate.
929      *
930      * @param other  The date being compared to.
931      *
932      * @return <code>true</code> if this SerialDate represents an earlier date
933      *         compared to the specified SerialDate.
934      */
935     public abstract boolean isBefore(SerialDate other);
936
937     /**
938      * Returns true if this SerialDate represents the same date as the
939      * specified SerialDate.
940      *
941      * @param other  the date being compared to.
942      *
943      * @return <code>true<code> if this SerialDate represents the same date
944      *         as the specified SerialDate.
945      */
946     public abstract boolean isOnOrBefore(SerialDate other);
947
948     /**
949      * Returns true if this SerialDate represents the same date as the
950      * specified SerialDate.
951      *
952      * @param other  the date being compared to.
953      *
954      * @return <code>true</code> if this SerialDate represents the same date
955      *         as the specified SerialDate.
956      */
957     public abstract boolean isAfter(SerialDate other);
958
959     /**
960      * Returns true if this SerialDate represents the same date as the
961      * specified SerialDate.
962      *
963      * @param other  the date being compared to.
964      *
965      * @return <code>true</code> if this SerialDate represents the same date
966      *         as the specified SerialDate.
967      */
968     public abstract boolean isOnOrAfter(SerialDate other);
969
```

```
970     /**
971      * Returns <code>true</code> if this {@link SerialDate} is within the
972      * specified range (INCLUSIVE).  The date order of d1 and d2 is not
973      * important.
974      *
975      * @param d1  a boundary date for the range.
976      * @param d2  the other boundary date for the range.
977      *
978      * @return A boolean.
979      */
980     public abstract boolean isInRange(SerialDate d1, SerialDate d2);
981
982     /**
983      * Returns <code>true</code> if this {@link SerialDate} is within the
984      * specified range (caller specifies whether or not the end-points are
985      * included).  The date order of d1 and d2 is not important.
986      *
987      * @param d1  a boundary date for the range.
988      * @param d2  the other boundary date for the range.
989      * @param include  a code that controls whether or not the start and end
990      *                 dates are included in the range.
991      *
992      * @return A boolean.
993      */
994     public abstract boolean isInRange(SerialDate d1, SerialDate d2,
995                                       int include);
996
997     /**
998      * Returns the latest date that falls on the specified day-of-the-week and
999      * is BEFORE this date.
1000     *
1001     * @param targetDOW  a code for the target day-of-the-week.
1002     *
1003     * @return the latest date that falls on the specified day-of-the-week and
1004     *         is BEFORE this date.
1005     */
1006    public SerialDate getPreviousDayOfWeek(final int targetDOW) {
1007        return getPreviousDayOfWeek(targetDOW, this);
1008    }
1009
1010    /**
1011     * Returns the earliest date that falls on the specified day-of-the-week
1012     * and is AFTER this date.
1013     *
1014     * @param targetDOW  a code for the target day-of-the-week.
1015     *
1016     * @return the earliest date that falls on the specified day-of-the-week
1017     *         and is AFTER this date.
1018     */
1019    public SerialDate getFollowingDayOfWeek(final int targetDOW) {
1020        return getFollowingDayOfWeek(targetDOW, this);
1021    }
1022
1023    /**
1024     * Returns the nearest date that falls on the specified day-of-the-week.
1025     *
1026     * @param targetDOW  a code for the target day-of-the-week.
1027     *
1028     * @return the nearest date that falls on the specified day-of-the-week.
1029     */
1030    public SerialDate getNearestDayOfWeek(final int targetDOW) {
1031        return getNearestDayOfWeek(targetDOW, this);
1032    }
1033
1034 }
```

Listing B-2

SerialDateTest.java

```
/* ===========================================================================
 * JCommon : a free general purpose class library for the Java(tm) platform
 * ===========================================================================
 *
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 *
 * Project Info:  http://www.jfree.org/jcommon/index.html
 *
 * This library is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This library 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 Lesser General Public
 * License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 * USA.
 *
 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
 * in the United States and other countries.]
 *
 * --------------------
 * SerialDateTests.java
 * --------------------
 * (C) Copyright 2001-2005, by Object Refinery Limited.
 *
 * Original Author:  David Gilbert (for Object Refinery Limited);
 * Contributor(s):   -;
 *
 * $Id: SerialDateTests.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
 *
 * Changes
 * -------
 * 15-Nov-2001 : Version 1 (DG);
 * 25-Jun-2002 : Removed unnecessary import (DG);
 * 24-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 * 13-Mar-2003 : Added serialization test (DG);
 * 05-Jan-2005 : Added test for bug report 1096282 (DG);
 *
 */

package org.jfree.date.junit;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

import org.jfree.date.MonthConstants;
import org.jfree.date.SerialDate;

```

```
 63 /**
 64  * Some JUnit tests for the {@link SerialDate} class.
 65  */
 66 public class SerialDateTests extends TestCase {
 67
 68     /** Date representing November 9. */
 69     private SerialDate nov9Y2001;
 70
 71     /**
 72      * Creates a new test case.
 73      *
 74      * @param name  the name.
 75      */
 76     public SerialDateTests(final String name) {
 77         super(name);
 78     }
 79
 80     /**
 81      * Returns a test suite for the JUnit test runner.
 82      *
 83      * @return The test suite.
 84      */
 85     public static Test suite() {
 86         return new TestSuite(SerialDateTests.class);
 87     }
 88
 89     /**
 90      * Problem set up.
 91      */
 92     protected void setUp() {
 93         this.nov9Y2001 = SerialDate.createInstance(9, MonthConstants.NOVEMBER, 2001);
 94     }
 95
 96     /**
 97      * 9 Nov 2001 plus two months should be 9 Jan 2002.
 98      */
 99     public void testAddMonthsTo9Nov2001() {
100         final SerialDate jan9Y2002 = SerialDate.addMonths(2, this.nov9Y2001);
101         final SerialDate answer = SerialDate.createInstance(9, 1, 2002);
102         assertEquals(answer, jan9Y2002);
103     }
104
105     /**
106      * A test case for a reported bug, now fixed.
107      */
108     public void testAddMonthsTo5Oct2003() {
109         final SerialDate d1 = SerialDate.createInstance(5, MonthConstants.OCTOBER, 2003);
110         final SerialDate d2 = SerialDate.addMonths(2, d1);
111         assertEquals(d2, SerialDate.createInstance(5, MonthConstants.DECEMBER, 2003));
112     }
113
114     /**
115      * A test case for a reported bug, now fixed.
116      */
117     public void testAddMonthsTo1Jan2003() {
118         final SerialDate d1 = SerialDate.createInstance(1, MonthConstants.JANUARY, 2003);
119         final SerialDate d2 = SerialDate.addMonths(0, d1);
120         assertEquals(d2, d1);
121     }
122
123     /**
124      * Monday preceding Friday 9 November 2001 should be 5 November.
```

```
125      */
126     public void testMondayPrecedingFriday9Nov2001() {
127         SerialDate mondayBefore = SerialDate.getPreviousDayOfWeek(
128             SerialDate.MONDAY, this.nov9Y2001
129         );
130         assertEquals(5, mondayBefore.getDayOfMonth());
131     }
132 
133     /**
134      * Monday following Friday 9 November 2001 should be 12 November.
135      */
136     public void testMondayFollowingFriday9Nov2001() {
137         SerialDate mondayAfter = SerialDate.getFollowingDayOfWeek(
138             SerialDate.MONDAY, this.nov9Y2001
139         );
140         assertEquals(12, mondayAfter.getDayOfMonth());
141     }
142 
143     /**
144      * Monday nearest Friday 9 November 2001 should be 12 November.
145      */
146     public void testMondayNearestFriday9Nov2001() {
147         SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(
148             SerialDate.MONDAY, this.nov9Y2001
149         );
150         assertEquals(12, mondayNearest.getDayOfMonth());
151     }
152 
153     /**
154      * The Monday nearest to 22nd January 1970 falls on the 19th.
155      */
156     public void testMondayNearest22Jan1970() {
157         SerialDate jan22Y1970 = SerialDate.createInstance(22, MonthConstants.JANUARY, 1970);
158         SerialDate mondayNearest=SerialDate.getNearestDayOfWeek(SerialDate.MONDAY, jan22Y1970);
159         assertEquals(19, mondayNearest.getDayOfMonth());
160     }
161 
162     /**
163      * Problem that the conversion of days to strings returns the right result.  Actually, this
164      * result depends on the Locale so this test needs to be modified.
165      */
166     public void testWeekdayCodeToString() {
167 
168         final String test = SerialDate.weekdayCodeToString(SerialDate.SATURDAY);
169         assertEquals("Saturday", test);
170 
171     }
172 
173     /**
174      * Test the conversion of a string to a weekday.  Note that this test will fail if the
175      * default locale doesn't use English weekday names...devise a better test!
176      */
177     public void testStringToWeekday() {
178 
179         int weekday = SerialDate.stringToWeekdayCode("Wednesday");
180         assertEquals(SerialDate.WEDNESDAY, weekday);
181 
182         weekday = SerialDate.stringToWeekdayCode(" Wednesday ");
183         assertEquals(SerialDate.WEDNESDAY, weekday);
184 
185         weekday = SerialDate.stringToWeekdayCode("Wed");
186         assertEquals(SerialDate.WEDNESDAY, weekday);
```

```
187
188     }
189
190     /**
191      * Test the conversion of a string to a month.  Note that this test will fail if the
192      * default locale doesn't use English month names...devise a better test!
193      */
194     public void testStringToMonthCode() {
195
196         int m = SerialDate.stringToMonthCode("January");
197         assertEquals(MonthConstants.JANUARY, m);
198
199         m = SerialDate.stringToMonthCode(" January ");
200         assertEquals(MonthConstants.JANUARY, m);
201
202         m = SerialDate.stringToMonthCode("Jan");
203         assertEquals(MonthConstants.JANUARY, m);
204
205     }
206
207     /**
208      * Tests the conversion of a month code to a string.
209      */
210     public void testMonthCodeToStringCode() {
211
212         final String test = SerialDate.monthCodeToString(MonthConstants.DECEMBER);
213         assertEquals("December", test);
214
215     }
216
217     /**
218      * 1900 is not a leap year.
219      */
220     public void testIsNotLeapYear1900() {
221         assertTrue(!SerialDate.isLeapYear(1900));
222     }
223
224     /**
225      * 2000 is a leap year.
226      */
227     public void testIsLeapYear2000() {
228         assertTrue(SerialDate.isLeapYear(2000));
229     }
230
231     /**
232      * The number of leap years from 1900 up-to-and-including 1899 is 0.
233      */
234     public void testLeapYearCount1899() {
235         assertEquals(SerialDate.leapYearCount(1899), 0);
236     }
237
238     /**
239      * The number of leap years from 1900 up-to-and-including 1903 is 0.
240      */
241     public void testLeapYearCount1903() {
242         assertEquals(SerialDate.leapYearCount(1903), 0);
243     }
244
245     /**
246      * The number of leap years from 1900 up-to-and-including 1904 is 1.
247      */
248     public void testLeapYearCount1904() {
249         assertEquals(SerialDate.leapYearCount(1904), 1);
```

```
250     }
251
252     /**
253      * The number of leap years from 1900 up-to-and-including 1999 is 24.
254      */
255     public void testLeapYearCount1999() {
256         assertEquals(SerialDate.leapYearCount(1999), 24);
257     }
258
259     /**
260      * The number of leap years from 1900 up-to-and-including 2000 is 25.
261      */
262     public void testLeapYearCount2000() {
263         assertEquals(SerialDate.leapYearCount(2000), 25);
264     }
265
266     /**
267      * Serialize an instance, restore it, and check for equality.
268      */
269     public void testSerialization() {
270
271         SerialDate d1 = SerialDate.createInstance(15, 4, 2000);
272         SerialDate d2 = null;
273
274         try {
275             ByteArrayOutputStream buffer = new ByteArrayOutputStream();
276             ObjectOutput out = new ObjectOutputStream(buffer);
277             out.writeObject(d1);
278             out.close();
279
280             ObjectInput in = new ObjectInputStream(
                                    new ByteArrayInputStream(buffer.toByteArray()));
281             d2 = (SerialDate) in.readObject();
282             in.close();
283         }
284         catch (Exception e) {
285             System.out.println(e.toString());
286         }
287         assertEquals(d1, d2);
288
289     }
290
291     /**
292      * A test for bug report 1096282 (now fixed).
293      */
294     public void test1096282() {
295         SerialDate d = SerialDate.createInstance(29, 2, 2004);
296         d = SerialDate.addYears(1, d);
297         SerialDate expected = SerialDate.createInstance(28, 2, 2005);
298         assertTrue(d.isOn(expected));
299     }
300
301     /**
302      * Miscellaneous tests for the addMonths() method.
303      */
304     public void testAddMonths() {
305         SerialDate d1 = SerialDate.createInstance(31, 5, 2004);
306
307         SerialDate d2 = SerialDate.addMonths(1, d1);
308         assertEquals(30, d2.getDayOfMonth());
309         assertEquals(6, d2.getMonth());
310         assertEquals(2004, d2.getYYYY());
```

```
311
312         SerialDate d3 = SerialDate.addMonths(2, d1);
313         assertEquals(31, d3.getDayOfMonth());
314         assertEquals(7, d3.getMonth());
315         assertEquals(2004, d3.getYYYY());
316
317         SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
318         assertEquals(30, d4.getDayOfMonth());
319         assertEquals(7, d4.getMonth());
320         assertEquals(2004, d4.getYYYY());
321     }
322 }
```

Listing B-3

`MonthConstants.java`

```
/* ===========================================================================
 * JCommon : a free general purpose class library for the Java(tm) platform
 * ===========================================================================
 *
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 *
 * Project Info:  http://www.jfree.org/jcommon/index.html
 *
 * This library is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This library 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 Lesser General Public
 * License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 * USA.
 *
 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
 * in the United States and other countries.]
 *
 * -------------------
 * MonthConstants.java
 * -------------------
 * (C) Copyright 2002, 2003, by Object Refinery Limited.
 *
 * Original Author:  David Gilbert (for Object Refinery Limited);
 * Contributor(s):   -;
 *
 * $Id: MonthConstants.java,v 1.4 2005/11/16 15:58:40 taqua Exp $
 *
 * Changes
 * -------
 * 29-May-2002 : Version 1 (code moved from SerialDate class) (DG);
 *
 */

package org.jfree.date;

/**
 * Useful constants for months.  Note that these are NOT equivalent to the
 * constants defined by java.util.Calendar (where JANUARY=0 and DECEMBER=11).
 * <P>
 * Used by the SerialDate and RegularTimePeriod classes.
 *
 * @author David Gilbert
 */
public interface MonthConstants {

    /** Constant for January. */
    public static final int JANUARY = 1;

    /** Constant for February. */
    public static final int FEBRUARY = 2;

    /** Constant for March. */
```

```
62      public static final int MARCH = 3;
63
64      /** Constant for April. */
65      public static final int APRIL = 4;
66
67      /** Constant for May. */
68      public static final int MAY = 5;
69
70      /** Constant for June. */
71      public static final int JUNE = 6;
72
73      /** Constant for July. */
74      public static final int JULY = 7;
75
76      /** Constant for August. */
77      public static final int AUGUST = 8;
78
79      /** Constant for September. */
80      public static final int SEPTEMBER = 9;
81
82      /** Constant for October. */
83      public static final int OCTOBER = 10;
84
85      /** Constant for November. */
86      public static final int NOVEMBER = 11;
87
88      /** Constant for December. */
89      public static final int DECEMBER = 12;
90
91  }
```

Listing B-4

BobsSerialDateTest.java

```
 1 package org.jfree.date.junit;
 2
 3 import junit.framework.TestCase;
 4 import org.jfree.date.*;
 5 import static org.jfree.date.SerialDate.*;
 6
 7 import java.util.*;
 8
 9 public class BobsSerialDateTest extends TestCase {
10
11   public void testIsValidWeekdayCode() throws Exception {
12     for (int day = 1; day <= 7; day++)
13       assertTrue(isValidWeekdayCode(day));
14     assertFalse(isValidWeekdayCode(0));
15     assertFalse(isValidWeekdayCode(8));
16   }
17
18   public void testStringToWeekdayCode() throws Exception {
19
20     assertEquals(-1, stringToWeekdayCode("Hello"));
21     assertEquals(MONDAY, stringToWeekdayCode("Monday"));
22     assertEquals(MONDAY, stringToWeekdayCode("Mon"));
23 //todo    assertEquals(MONDAY,stringToWeekdayCode("monday"));
24 //     assertEquals(MONDAY,stringToWeekdayCode("MONDAY"));
25 //     assertEquals(MONDAY, stringToWeekdayCode("mon"));
26
27     assertEquals(TUESDAY, stringToWeekdayCode("Tuesday"));
28     assertEquals(TUESDAY, stringToWeekdayCode("Tue"));
29 //     assertEquals(TUESDAY,stringToWeekdayCode("tuesday"));
30 //     assertEquals(TUESDAY,stringToWeekdayCode("TUESDAY"));
31 //     assertEquals(TUESDAY, stringToWeekdayCode("tue"));
32 //     assertEquals(TUESDAY, stringToWeekdayCode("tues"));
33
34     assertEquals(WEDNESDAY, stringToWeekdayCode("Wednesday"));
35     assertEquals(WEDNESDAY, stringToWeekdayCode("Wed"));
36 //     assertEquals(WEDNESDAY,stringToWeekdayCode("wednesday"));
37 //     assertEquals(WEDNESDAY,stringToWeekdayCode("WEDNESDAY"));
38 //     assertEquals(WEDNESDAY, stringToWeekdayCode("wed"));
39
40     assertEquals(THURSDAY, stringToWeekdayCode("Thursday"));
41     assertEquals(THURSDAY, stringToWeekdayCode("Thu"));
42 //     assertEquals(THURSDAY,stringToWeekdayCode("thursday"));
43 //     assertEquals(THURSDAY,stringToWeekdayCode("THURSDAY"));
44 //     assertEquals(THURSDAY, stringToWeekdayCode("thu"));
45 //     assertEquals(THURSDAY, stringToWeekdayCode("thurs"));
46
47     assertEquals(FRIDAY, stringToWeekdayCode("Friday"));
48     assertEquals(FRIDAY, stringToWeekdayCode("Fri"));
49 //     assertEquals(FRIDAY,stringToWeekdayCode("friday"));
50 //     assertEquals(FRIDAY,stringToWeekdayCode("FRIDAY"));
51 //     assertEquals(FRIDAY, stringToWeekdayCode("fri"));
52
53     assertEquals(SATURDAY, stringToWeekdayCode("Saturday"));
54     assertEquals(SATURDAY, stringToWeekdayCode("Sat"));
55 //     assertEquals(SATURDAY,stringToWeekdayCode("saturday"));
56 //     assertEquals(SATURDAY,stringToWeekdayCode("SATURDAY"));
57 //     assertEquals(SATURDAY, stringToWeekdayCode("sat"));
58
59     assertEquals(SUNDAY, stringToWeekdayCode("Sunday"));
60     assertEquals(SUNDAY, stringToWeekdayCode("Sun"));
61 //     assertEquals(SUNDAY,stringToWeekdayCode("sunday"));
62 //     assertEquals(SUNDAY,stringToWeekdayCode("SUNDAY"));
```

```
63 //    assertEquals(SUNDAY, stringToWeekdayCode("sun"));
64   }
65
66   public void testWeekdayCodeToString() throws Exception {
67     assertEquals("Sunday", weekdayCodeToString(SUNDAY));
68     assertEquals("Monday", weekdayCodeToString(MONDAY));
69     assertEquals("Tuesday", weekdayCodeToString(TUESDAY));
70     assertEquals("Wednesday", weekdayCodeToString(WEDNESDAY));
71     assertEquals("Thursday", weekdayCodeToString(THURSDAY));
72     assertEquals("Friday", weekdayCodeToString(FRIDAY));
73     assertEquals("Saturday", weekdayCodeToString(SATURDAY));
74   }
75
76   public void testIsValidMonthCode() throws Exception {
77     for (int i = 1; i <= 12; i++)
78       assertTrue(isValidMonthCode(i));
79     assertFalse(isValidMonthCode(0));
80     assertFalse(isValidMonthCode(13));
81   }
82
83   public void testMonthToQuarter() throws Exception {
84     assertEquals(1, monthCodeToQuarter(JANUARY));
85     assertEquals(1, monthCodeToQuarter(FEBRUARY));
86     assertEquals(1, monthCodeToQuarter(MARCH));
87     assertEquals(2, monthCodeToQuarter(APRIL));
88     assertEquals(2, monthCodeToQuarter(MAY));
89     assertEquals(2, monthCodeToQuarter(JUNE));
90     assertEquals(3, monthCodeToQuarter(JULY));
91     assertEquals(3, monthCodeToQuarter(AUGUST));
92     assertEquals(3, monthCodeToQuarter(SEPTEMBER));
93     assertEquals(4, monthCodeToQuarter(OCTOBER));
94     assertEquals(4, monthCodeToQuarter(NOVEMBER));
95     assertEquals(4, monthCodeToQuarter(DECEMBER));
96
97     try {
98       monthCodeToQuarter(-1);
99       fail("Invalid Month Code should throw exception");
100    } catch (IllegalArgumentException e) {
101    }
102  }
103
104  public void testMonthCodeToString() throws Exception {
105    assertEquals("January", monthCodeToString(JANUARY));
106    assertEquals("February", monthCodeToString(FEBRUARY));
107    assertEquals("March", monthCodeToString(MARCH));
108    assertEquals("April", monthCodeToString(APRIL));
109    assertEquals("May", monthCodeToString(MAY));
110    assertEquals("June", monthCodeToString(JUNE));
111    assertEquals("July", monthCodeToString(JULY));
112    assertEquals("August", monthCodeToString(AUGUST));
113    assertEquals("September", monthCodeToString(SEPTEMBER));
114    assertEquals("October", monthCodeToString(OCTOBER));
115    assertEquals("November", monthCodeToString(NOVEMBER));
116    assertEquals("December", monthCodeToString(DECEMBER));
117
118    assertEquals("Jan", monthCodeToString(JANUARY, true));
119    assertEquals("Feb", monthCodeToString(FEBRUARY, true));
120    assertEquals("Mar", monthCodeToString(MARCH, true));
121    assertEquals("Apr", monthCodeToString(APRIL, true));
122    assertEquals("May", monthCodeToString(MAY, true));
123    assertEquals("Jun", monthCodeToString(JUNE, true));
124    assertEquals("Jul", monthCodeToString(JULY, true));
```

```
125     assertEquals("Aug", monthCodeToString(AUGUST, true));
126     assertEquals("Sep", monthCodeToString(SEPTEMBER, true));
127     assertEquals("Oct", monthCodeToString(OCTOBER, true));
128     assertEquals("Nov", monthCodeToString(NOVEMBER, true));
129     assertEquals("Dec", monthCodeToString(DECEMBER, true));
130
131     try {
132       monthCodeToString(-1);
133       fail("Invalid month code should throw exception");
134     } catch (IllegalArgumentException e) {
135     }
136
137   }
138
139   public void testStringToMonthCode() throws Exception {
140     assertEquals(JANUARY,stringToMonthCode("1"));
141     assertEquals(FEBRUARY,stringToMonthCode("2"));
142     assertEquals(MARCH,stringToMonthCode("3"));
143     assertEquals(APRIL,stringToMonthCode("4"));
144     assertEquals(MAY,stringToMonthCode("5"));
145     assertEquals(JUNE,stringToMonthCode("6"));
146     assertEquals(JULY,stringToMonthCode("7"));
147     assertEquals(AUGUST,stringToMonthCode("8"));
148     assertEquals(SEPTEMBER,stringToMonthCode("9"));
149     assertEquals(OCTOBER,stringToMonthCode("10"));
150     assertEquals(NOVEMBER, stringToMonthCode("11"));
151     assertEquals(DECEMBER,stringToMonthCode("12"));
152
153 //todo    assertEquals(-1, stringToMonthCode("0"));
154 //    assertEquals(-1, stringToMonthCode("13"));
155
156     assertEquals(-1,stringToMonthCode("Hello"));
157
158     for (int m = 1; m <= 12; m++) {
159       assertEquals(m, stringToMonthCode(monthCodeToString(m, false)));
160       assertEquals(m, stringToMonthCode(monthCodeToString(m, true)));
161     }
162
163 //    assertEquals(1,stringToMonthCode("jan"));
164 //    assertEquals(2,stringToMonthCode("feb"));
165 //    assertEquals(3,stringToMonthCode("mar"));
166 //    assertEquals(4,stringToMonthCode("apr"));
167 //    assertEquals(5,stringToMonthCode("may"));
168 //    assertEquals(6,stringToMonthCode("jun"));
169 //    assertEquals(7,stringToMonthCode("jul"));
170 //    assertEquals(8,stringToMonthCode("aug"));
171 //    assertEquals(9,stringToMonthCode("sep"));
172 //    assertEquals(10,stringToMonthCode("oct"));
173 //    assertEquals(11,stringToMonthCode("nov"));
174 //    assertEquals(12,stringToMonthCode("dec"));
175
176 //    assertEquals(1,stringToMonthCode("JAN"));
177 //    assertEquals(2,stringToMonthCode("FEB"));
178 //    assertEquals(3,stringToMonthCode("MAR"));
179 //    assertEquals(4,stringToMonthCode("APR"));
180 //    assertEquals(5,stringToMonthCode("MAY"));
181 //    assertEquals(6,stringToMonthCode("JUN"));
182 //    assertEquals(7,stringToMonthCode("JUL"));
183 //    assertEquals(8,stringToMonthCode("AUG"));
184 //    assertEquals(9,stringToMonthCode("SEP"));
185 //    assertEquals(10,stringToMonthCode("OCT"));
186 //    assertEquals(11,stringToMonthCode("NOV"));
187 //    assertEquals(12,stringToMonthCode("DEC"));
188
189 //    assertEquals(1,stringToMonthCode("january"));
```

```
190 //    assertEquals(2,stringToMonthCode("february"));
191 //    assertEquals(3,stringToMonthCode("march"));
192 //    assertEquals(4,stringToMonthCode("april"));
193 //    assertEquals(5,stringToMonthCode("may"));
194 //    assertEquals(6,stringToMonthCode("june"));
195 //    assertEquals(7,stringToMonthCode("july"));
196 //    assertEquals(8,stringToMonthCode("august"));
197 //    assertEquals(9,stringToMonthCode("september"));
198 //    assertEquals(10,stringToMonthCode("october"));
199 //    assertEquals(11,stringToMonthCode("november"));
200 //    assertEquals(12,stringToMonthCode("december"));
201
202 //    assertEquals(1,stringToMonthCode("JANUARY"));
203 //    assertEquals(2,stringToMonthCode("FEBRUARY"));
204 //    assertEquals(3,stringToMonthCode("MAR"));
205 //    assertEquals(4,stringToMonthCode("APRIL"));
206 //    assertEquals(5,stringToMonthCode("MAY"));
207 //    assertEquals(6,stringToMonthCode("JUNE"));
208 //    assertEquals(7,stringToMonthCode("JULY"));
209 //    assertEquals(8,stringToMonthCode("AUGUST"));
210 //    assertEquals(9,stringToMonthCode("SEPTEMBER"));
211 //    assertEquals(10,stringToMonthCode("OCTOBER"));
212 //    assertEquals(11,stringToMonthCode("NOVEMBER"));
213 //    assertEquals(12,stringToMonthCode("DECEMBER"));
214   }
215
216   public void testIsValidWeekInMonthCode() throws Exception {
217     for (int w = 0; w <= 4; w++) {
218       assertTrue(isValidWeekInMonthCode(w));
219     }
220     assertFalse(isValidWeekInMonthCode(5));
221   }
222
223   public void testIsLeapYear() throws Exception {
224     assertFalse(isLeapYear(1900));
225     assertFalse(isLeapYear(1901));
226     assertFalse(isLeapYear(1902));
227     assertFalse(isLeapYear(1903));
228     assertTrue(isLeapYear(1904));
229     assertTrue(isLeapYear(1908));
230     assertFalse(isLeapYear(1955));
231     assertTrue(isLeapYear(1964));
232     assertTrue(isLeapYear(1980));
233     assertTrue(isLeapYear(2000));
234     assertFalse(isLeapYear(2001));
235     assertFalse(isLeapYear(2100));
236   }
237
238   public void testLeapYearCount() throws Exception {
239     assertEquals(0, leapYearCount(1900));
240     assertEquals(0, leapYearCount(1901));
241     assertEquals(0, leapYearCount(1902));
242     assertEquals(0, leapYearCount(1903));
243     assertEquals(1, leapYearCount(1904));
244     assertEquals(1, leapYearCount(1905));
245     assertEquals(1, leapYearCount(1906));
246     assertEquals(1, leapYearCount(1907));
247     assertEquals(2, leapYearCount(1908));
248     assertEquals(24, leapYearCount(1999));
249     assertEquals(25, leapYearCount(2001));
250     assertEquals(49, leapYearCount(2101));
```

```
251     assertEquals(73, leapYearCount(2201));
252     assertEquals(97, leapYearCount(2301));
253     assertEquals(122, leapYearCount(2401));
254   }
255
256   public void testLastDayOfMonth() throws Exception {
257     assertEquals(31, lastDayOfMonth(JANUARY, 1901));
258     assertEquals(28, lastDayOfMonth(FEBRUARY, 1901));
259     assertEquals(31, lastDayOfMonth(MARCH, 1901));
260     assertEquals(30, lastDayOfMonth(APRIL, 1901));
261     assertEquals(31, lastDayOfMonth(MAY, 1901));
262     assertEquals(30, lastDayOfMonth(JUNE, 1901));
263     assertEquals(31, lastDayOfMonth(JULY, 1901));
264     assertEquals(31, lastDayOfMonth(AUGUST, 1901));
265     assertEquals(30, lastDayOfMonth(SEPTEMBER, 1901));
266     assertEquals(31, lastDayOfMonth(OCTOBER, 1901));
267     assertEquals(30, lastDayOfMonth(NOVEMBER, 1901));
268     assertEquals(31, lastDayOfMonth(DECEMBER, 1901));
269     assertEquals(29, lastDayOfMonth(FEBRUARY, 1904));
270   }
271
272   public void testAddDays() throws Exception {
273     SerialDate newYears = d(1, JANUARY, 1900);
274     assertEquals(d(2, JANUARY, 1900), addDays(1, newYears));
275     assertEquals(d(1, FEBRUARY, 1900), addDays(31, newYears));
276     assertEquals(d(1, JANUARY, 1901), addDays(365, newYears));
277     assertEquals(d(31, DECEMBER, 1904), addDays(5 * 365, newYears));
278   }
279
280   private static SpreadsheetDate d(int day, int month, int year) {return new
SpreadsheetDate(day, month, year);}
281
282   public void testAddMonths() throws Exception {
283     assertEquals(d(1, FEBRUARY, 1900), addMonths(1, d(1, JANUARY, 1900)));
284     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(31, JANUARY, 1900)));
285     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(30, JANUARY, 1900)));
286     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(29, JANUARY, 1900)));
287     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(28, JANUARY, 1900)));
288     assertEquals(d(27, FEBRUARY, 1900), addMonths(1, d(27, JANUARY, 1900)));
289
290     assertEquals(d(30, JUNE, 1900), addMonths(5, d(31, JANUARY, 1900)));
291     assertEquals(d(30, JUNE, 1901), addMonths(17, d(31, JANUARY, 1900)));
292
293     assertEquals(d(29, FEBRUARY, 1904), addMonths(49, d(31, JANUARY, 1900)));
294
295   }
296
297   public void testAddYears() throws Exception {
298     assertEquals(d(1, JANUARY, 1901), addYears(1, d(1, JANUARY, 1900)));
299     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(29, FEBRUARY, 1904)));
300     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(28, FEBRUARY, 1904)));
301     assertEquals(d(28, FEBRUARY, 1904), addYears(1, d(28, FEBRUARY, 1903)));
302   }
303
304   public void testGetPreviousDayOfWeek() throws Exception {
305     assertEquals(d(24, FEBRUARY, 2006), getPreviousDayOfWeek(FRIDAY, d(1, MARCH, 2006)));
306     assertEquals(d(22, FEBRUARY, 2006), getPreviousDayOfWeek(WEDNESDAY, d(1, MARCH, 2006)));
307     assertEquals(d(29, FEBRUARY, 2004), getPreviousDayOfWeek(SUNDAY, d(3, MARCH, 2004)));
308     assertEquals(d(29, DECEMBER, 2004), getPreviousDayOfWeek(WEDNESDAY, d(5, JANUARY, 2005)));
309
310     try {
311       getPreviousDayOfWeek(-1, d(1, JANUARY, 2006));
312       fail("Invalid day of week code should throw exception");
313     } catch (IllegalArgumentException e) {
314     }
```

```
315   }
316
317   public void testGetFollowingDayOfWeek() throws Exception {
318 //    assertEquals(d(1, JANUARY, 2005),getFollowingDayOfWeek(SATURDAY, d(25, DECEMBER, 2004)));
319     assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(26, DECEMBER, 2004)));
320     assertEquals(d(3, MARCH, 2004), getFollowingDayOfWeek(WEDNESDAY, d(28, FEBRUARY, 2004)));
321
322     try {
323       getFollowingDayOfWeek(-1, d(1, JANUARY, 2006));
324       fail("Invalid day of week code should throw exception");
325     } catch (IllegalArgumentException e) {
326     }
327   }
328
329   public void testGetNearestDayOfWeek() throws Exception {
330     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(16, APRIL, 2006)));
331     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(17, APRIL, 2006)));
332     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(18, APRIL, 2006)));
333     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(19, APRIL, 2006)));
334     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(20, APRIL, 2006)));
335     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(21, APRIL, 2006)));
336     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(22, APRIL, 2006)));
337
338 //todo    assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(16, APRIL, 2006)));
339     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(17, APRIL, 2006)));
340     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(18, APRIL, 2006)));
341     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(19, APRIL, 2006)));
342     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(20, APRIL, 2006)));
343     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(21, APRIL, 2006)));
344     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(22, APRIL, 2006)));
345
346 //    assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(16, APRIL, 2006)));
347 //    assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(17, APRIL, 2006)));
348     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(18, APRIL, 2006)));
349     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(19, APRIL, 2006)));
350     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(20, APRIL, 2006)));
351     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(21, APRIL, 2006)));
352     assertEquals(d(25, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(22, APRIL, 2006)));
353
354 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(16, APRIL, 2006)));
355 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(17, APRIL, 2006)));
356 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(18, APRIL, 2006)));
357     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(19, APRIL, 2006)));
358     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(20, APRIL, 2006)));
359     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(21, APRIL, 2006)));
360     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(22, APRIL, 2006)));
361
362 //    assertEquals(d(13, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(16, APRIL, 2006)));
363 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(17, APRIL, 2006)));
364 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(18, APRIL, 2006)));
365 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(19, APRIL, 2006)));
366     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(20, APRIL, 2006)));
367     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(21, APRIL, 2006)));
368     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(22, APRIL, 2006)));
369
370 //    assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(16, APRIL, 2006)));
371 //    assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(17, APRIL, 2006)));
372 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(18, APRIL, 2006)));
373 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(19, APRIL, 2006)));
374 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(20, APRIL, 2006)));
375     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(21, APRIL, 2006)));
376     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(22, APRIL, 2006)));
```

```
377
378 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(16, APRIL, 2006)));
379 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(17, APRIL, 2006)));
380 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(18, APRIL, 2006)));
381 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(19, APRIL, 2006)));
382 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(20, APRIL, 2006)));
383 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(21, APRIL, 2006)));
384     assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(22, APRIL, 2006)));
385
386     try {
387       getNearestDayOfWeek(-1, d(1, JANUARY, 2006));
388       fail("Invalid day of week code should throw exception");
389     } catch (IllegalArgumentException e) {
390     }
391   }
392
393   public void testEndOfCurrentMonth() throws Exception {
394     SerialDate d = SerialDate.createInstance(2);
395     assertEquals(d(31, JANUARY, 2006), d.getEndOfCurrentMonth(d(1, JANUARY, 2006)));
396     assertEquals(d(28, FEBRUARY, 2006), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2006)));
397     assertEquals(d(31, MARCH, 2006), d.getEndOfCurrentMonth(d(1, MARCH, 2006)));
398     assertEquals(d(30, APRIL, 2006), d.getEndOfCurrentMonth(d(1, APRIL, 2006)));
399     assertEquals(d(31, MAY, 2006), d.getEndOfCurrentMonth(d(1, MAY, 2006)));
400     assertEquals(d(30, JUNE, 2006), d.getEndOfCurrentMonth(d(1, JUNE, 2006)));
401     assertEquals(d(31, JULY, 2006), d.getEndOfCurrentMonth(d(1, JULY, 2006)));
402     assertEquals(d(31, AUGUST, 2006), d.getEndOfCurrentMonth(d(1, AUGUST, 2006)));
403     assertEquals(d(30, SEPTEMBER, 2006), d.getEndOfCurrentMonth(d(1, SEPTEMBER, 2006)));
404     assertEquals(d(31, OCTOBER, 2006), d.getEndOfCurrentMonth(d(1, OCTOBER, 2006)));
405     assertEquals(d(30, NOVEMBER, 2006), d.getEndOfCurrentMonth(d(1, NOVEMBER, 2006)));
406     assertEquals(d(31, DECEMBER, 2006), d.getEndOfCurrentMonth(d(1, DECEMBER, 2006)));
407     assertEquals(d(29, FEBRUARY, 2008), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2008)));
408   }
409
410   public void testWeekInMonthToString() throws Exception {
411     assertEquals("First",weekInMonthToString(FIRST_WEEK_IN_MONTH));
412     assertEquals("Second",weekInMonthToString(SECOND_WEEK_IN_MONTH));
413     assertEquals("Third",weekInMonthToString(THIRD_WEEK_IN_MONTH));
414     assertEquals("Fourth",weekInMonthToString(FOURTH_WEEK_IN_MONTH));
415     assertEquals("Last",weekInMonthToString(LAST_WEEK_IN_MONTH));
416
417 //todo    try {
418 //      weekInMonthToString(-1);
419 //      fail("Invalid week code should throw exception");
420 //    } catch (IllegalArgumentException e) {
421 //    }
422   }
423
424   public void testRelativeToString() throws Exception {
425     assertEquals("Preceding",relativeToString(PRECEDING));
426     assertEquals("Nearest",relativeToString(NEAREST));
427     assertEquals("Following",relativeToString(FOLLOWING));
428
429 //todo    try {
430 //      relativeToString(-1000);
431 //      fail("Invalid relative code should throw exception");
432 //    } catch (IllegalArgumentException e) {
433 //    }
434   }
435
436   public void testCreateInstanceFromDDMMYYY() throws Exception {
437     SerialDate date = createInstance(1, JANUARY, 1900);
438     assertEquals(1,date.getDayOfMonth());
```

```
439     assertEquals(JANUARY,date.getMonth());
440     assertEquals(1900,date.getYYYY());
441     assertEquals(2,date.toSerial());
442   }
443
444   public void testCreateInstanceFromSerial() throws Exception {
445     assertEquals(d(1, JANUARY, 1900),createInstance(2));
446     assertEquals(d(1, JANUARY, 1901), createInstance(367));
447   }
448
449   public void testCreateInstanceFromJavaDate() throws Exception {
450     assertEquals(d(1, JANUARY, 1900),
                   createInstance(new GregorianCalendar(1900,0,1).getTime()));
451     assertEquals(d(1, JANUARY, 2006),
                   createInstance(new GregorianCalendar(2006,0,1).getTime()));
452   }
453
454   public static void main(String[] args) {
455     junit.textui.TestRunner.run(BobsSerialDateTest.class);
456   }
457 }
```

Listing B-5

SpreadsheetDate.java

```
/* ===========================================================================
 * JCommon : a free general purpose class library for the Java(tm) platform
 * ===========================================================================
 *
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 *
 * Project Info:  http://www.jfree.org/jcommon/index.html
 *
 * This library is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This library 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 Lesser General Public
 * License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 * USA.
 *
 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
 * in the United States and other countries.]
 *
 * --------------------
 * SpreadsheetDate.java
 * --------------------
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 *
 * Original Author:  David Gilbert (for Object Refinery Limited);
 * Contributor(s):   -;
 *
 * $Id: SpreadsheetDate.java,v 1.8 2005/11/03 09:25:39 mungady Exp $
 *
 * Changes
 * -------
 * 11-Oct-2001 : Version 1 (DG);
 * 05-Nov-2001 : Added getDescription() and setDescription() methods (DG);
 * 12-Nov-2001 : Changed name from ExcelDate.java to SpreadsheetDate.java (DG);
 *               Fixed a bug in calculating day, month and year from serial
 *               number (DG);
 * 24-Jan-2002 : Fixed a bug in calculating the serial number from the day,
 *               month and year.  Thanks to Trevor Hills for the report (DG);
 * 29-May-2002 : Added equals(Object) method (SourceForge ID 558850) (DG);
 * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 * 13-Mar-2003 : Implemented Serializable (DG);
 * 04-Sep-2003 : Completed isInRange() methods (DG);
 * 05-Sep-2003 : Implemented Comparable (DG);
 * 21-Oct-2003 : Added hashCode() method (DG);
 *
 */

package org.jfree.date;

import java.util.Calendar;
import java.util.Date;

/**
 * Represents a date using an integer, in a similar fashion to the
 * implementation in Microsoft Excel.  The range of dates supported is
 * 1-Jan-1900 to 31-Dec-9999.
 * <P>
```

```
 * Be aware that there is a deliberate bug in Excel that recognises the year
 * 1900 as a leap year when in fact it is not a leap year. You can find more
 * information on the Microsoft website in article Q181370:
 * <P>
 * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
 * <P>
 * Excel uses the convention that 1-Jan-1900 = 1.  This class uses the
 * convention 1-Jan-1900 = 2.
 * The result is that the day number in this class will be different to the
 * Excel figure for January and February 1900...but then Excel adds in an extra
 * day (29-Feb-1900 which does not actually exist!) and from that point forward
 * the day numbers will match.
 *
 * @author David Gilbert
 */
public class SpreadsheetDate extends SerialDate {

    /** For serialization. */
    private static final long serialVersionUID = -2039586705374454461L;

    /**
     * The day number (1-Jan-1900 = 2, 2-Jan-1900 = 3, ..., 31-Dec-9999 =
     * 2958465).
     */
    private int serial;

    /** The day of the month (1 to 28, 29, 30 or 31 depending on the month). */
    private int day;

    /** The month of the year (1 to 12). */
    private int month;

    /** The year (1900 to 9999). */
    private int year;

    /** An optional description for the date. */
    private String description;

    /**
     * Creates a new date instance.
     *
     * @param day  the day (in the range 1 to 28/29/30/31).
     * @param month  the month (in the range 1 to 12).
     * @param year  the year (in the range 1900 to 9999).
     */
    public SpreadsheetDate(final int day, final int month, final int year) {

        if ((year >= 1900) && (year <= 9999)) {
            this.year = year;
        }
        else {
            throw new IllegalArgumentException(
                "The 'year' argument must be in range 1900 to 9999."
            );
        }

        if ((month >= MonthConstants.JANUARY)
                && (month <= MonthConstants.DECEMBER)) {
            this.month = month;
        }
        else {
            throw new IllegalArgumentException(
                "The 'month' argument must be in the range 1 to 12."
            );
        }
```

```
130
131         if ((day >= 1) && (day <= SerialDate.lastDayOfMonth(month, year))) {
132             this.day = day;
133         }
134         else {
135             throw new IllegalArgumentException("Invalid 'day' argument.");
136         }
137
138         // the serial number needs to be synchronised with the day-month-year...
139         this.serial = calcSerial(day, month, year);
140
141         this.description = null;
142
143     }
144
145     /**
146      * Standard constructor - creates a new date object representing the
147      * specified day number (which should be in the range 2 to 2958465.
148      *
149      * @param serial  the serial number for the day (range: 2 to 2958465).
150      */
151     public SpreadsheetDate(final int serial) {
152
153         if ((serial >= SERIAL_LOWER_BOUND) && (serial <= SERIAL_UPPER_BOUND)) {
154             this.serial = serial;
155         }
156         else {
157             throw new IllegalArgumentException(
158                 "SpreadsheetDate: Serial must be in range 2 to 2958465.");
159         }
160
161         // the day-month-year needs to be synchronised with the serial number...
162         calcDayMonthYear();
163
164     }
165
166     /**
167      * Returns the description that is attached to the date.  It is not
168      * required that a date have a description, but for some applications it
169      * is useful.
170      *
171      * @return The description that is attached to the date.
172      */
173     public String getDescription() {
174         return this.description;
175     }
176
177     /**
178      * Sets the description for the date.
179      *
180      * @param description  the description for this date (<code>null</code>
181      *                     permitted).
182      */
183     public void setDescription(final String description) {
184         this.description = description;
185     }
186
187     /**
188      * Returns the serial number for the date, where 1 January 1900 = 2
189      * (this corresponds, almost, to the numbering system used in Microsoft
190      * Excel for Windows and Lotus 1-2-3).
191      *
```

```
192      * @return The serial number of this date.
193      */
194     public int toSerial() {
195         return this.serial;
196     }
197
198     /**
199      * Returns a <code>java.util.Date</code> equivalent to this date.
200      *
201      * @return The date.
202      */
203     public Date toDate() {
204         final Calendar calendar = Calendar.getInstance();
205         calendar.set(getYYYY(), getMonth() - 1, getDayOfMonth(), 0, 0, 0);
206         return calendar.getTime();
207     }
208
209     /**
210      * Returns the year (assume a valid range of 1900 to 9999).
211      *
212      * @return The year.
213      */
214     public int getYYYY() {
215         return this.year;
216     }
217
218     /**
219      * Returns the month (January = 1, February = 2, March = 3).
220      *
221      * @return The month of the year.
222      */
223     public int getMonth() {
224         return this.month;
225     }
226
227     /**
228      * Returns the day of the month.
229      *
230      * @return The day of the month.
231      */
232     public int getDayOfMonth() {
233         return this.day;
234     }
235
236     /**
237      * Returns a code representing the day of the week.
238      * <P>
239      * The codes are defined in the {@link SerialDate} class as:
240      * <code>SUNDAY</code>, <code>MONDAY</code>, <code>TUESDAY</code>,
241      * <code>WEDNESDAY</code>, <code>THURSDAY</code>, <code>FRIDAY</code>, and
242      * <code>SATURDAY</code>.
243      *
244      * @return A code representing the day of the week.
245      */
246     public int getDayOfWeek() {
247         return (this.serial + 6) % 7 + 1;
248     }
249
250     /**
251      * Tests the equality of this date with an arbitrary object.
252      * <P>
253      * This method will return true ONLY if the object is an instance of the
```

```
254      * {@link SerialDate} base class, and it represents the same day as this
255      * {@link SpreadsheetDate}.
256      *
257      * @param object  the object to compare (<code>null</code> permitted).
258      *
259      * @return A boolean.
260      */
261     public boolean equals(final Object object) {
262
263         if (object instanceof SerialDate) {
264             final SerialDate s = (SerialDate) object;
265             return (s.toSerial() == this.toSerial());
266         }
267         else {
268             return false;
269         }
270
271     }
272
273     /**
274      * Returns a hash code for this object instance.
275      *
276      * @return A hash code.
277      */
278     public int hashCode() {
279         return toSerial();
280     }
281
282     /**
283      * Returns the difference (in days) between this date and the specified
284      * 'other' date.
285      *
286      * @param other  the date being compared to.
287      *
288      * @return The difference (in days) between this date and the specified
289      *         'other' date.
290      */
291     public int compare(final SerialDate other) {
292         return this.serial - other.toSerial();
293     }
294
295     /**
296      * Implements the method required by the Comparable interface.
297      *
298      * @param other  the other object (usually another SerialDate).
299      *
300      * @return A negative integer, zero, or a positive integer as this object
301      *         is less than, equal to, or greater than the specified object.
302      */
303     public int compareTo(final Object other) {
304         return compare((SerialDate) other);
305     }
306
307     /**
308      * Returns true if this SerialDate represents the same date as the
309      * specified SerialDate.
310      *
311      * @param other  the date being compared to.
312      *
313      * @return <code>true</code> if this SerialDate represents the same date as
314      *         the specified SerialDate.
315      */
```

```
316     public boolean isOn(final SerialDate other) {
317         return (this.serial == other.toSerial());
318     }
319
320     /**
321      * Returns true if this SerialDate represents an earlier date compared to
322      * the specified SerialDate.
323      *
324      * @param other  the date being compared to.
325      *
326      * @return <code>true</code> if this SerialDate represents an earlier date
327      *         compared to the specified SerialDate.
328      */
329     public boolean isBefore(final SerialDate other) {
330         return (this.serial < other.toSerial());
331     }
332
333     /**
334      * Returns true if this SerialDate represents the same date as the
335      * specified SerialDate.
336      *
337      * @param other  the date being compared to.
338      *
339      * @return <code>true</code> if this SerialDate represents the same date
340      *         as the specified SerialDate.
341      */
342     public boolean isOnOrBefore(final SerialDate other) {
343         return (this.serial <= other.toSerial());
344     }
345
346     /**
347      * Returns true if this SerialDate represents the same date as the
348      * specified SerialDate.
349      *
350      * @param other  the date being compared to.
351      *
352      * @return <code>true</code> if this SerialDate represents the same date
353      *         as the specified SerialDate.
354      */
355     public boolean isAfter(final SerialDate other) {
356         return (this.serial > other.toSerial());
357     }
358
359     /**
360      * Returns true if this SerialDate represents the same date as the
361      * specified SerialDate.
362      *
363      * @param other  the date being compared to.
364      *
365      * @return <code>true</code> if this SerialDate represents the same date as
366      *         the specified SerialDate.
367      */
368     public boolean isOnOrAfter(final SerialDate other) {
369         return (this.serial >= other.toSerial());
370     }
371
372     /**
373      * Returns <code>true</code> if this {@link SerialDate} is within the
374      * specified range (INCLUSIVE).  The date order of d1 and d2 is not
375      * important.
376      *
377      * @param d1  a boundary date for the range.
378      * @param d2  the other boundary date for the range.
```

```
379      *
380      * @return A boolean.
381      */
382     public boolean isInRange(final SerialDate d1, final SerialDate d2) {
383         return isInRange(d1, d2, SerialDate.INCLUDE_BOTH);
384     }
385
386     /**
387      * Returns true if this SerialDate is within the specified range (caller
388      * specifies whether or not the end-points are included).  The order of d1
389      * and d2 is not important.
390      *
391      * @param d1  one boundary date for the range.
392      * @param d2  a second boundary date for the range.
393      * @param include  a code that controls whether or not the start and end
394      *                 dates are included in the range.
395      *
396      * @return <code>true</code> if this SerialDate is within the specified
397      *         range.
398      */
399     public boolean isInRange(final SerialDate d1, final SerialDate d2,
400                              final int include) {
401         final int s1 = d1.toSerial();
402         final int s2 = d2.toSerial();
403         final int start = Math.min(s1, s2);
404         final int end = Math.max(s1, s2);
405
406         final int s = toSerial();
407         if (include == SerialDate.INCLUDE_BOTH) {
408             return (s >= start && s <= end);
409         }
410         else if (include == SerialDate.INCLUDE_FIRST) {
411             return (s >= start && s < end);
412         }
413         else if (include == SerialDate.INCLUDE_SECOND) {
414             return (s > start && s <= end);
415         }
416         else {
417             return (s > start && s < end);
418         }
419     }
420
421     /**
422      * Calculate the serial number from the day, month and year.
423      * <P>
424      * 1-Jan-1900 = 2.
425      *
426      * @param d  the day.
427      * @param m  the month.
428      * @param y  the year.
429      *
430      * @return the serial number from the day, month and year.
431      */
432     private int calcSerial(final int d, final int m, final int y) {
433         final int yy = ((y - 1900) * 365) + SerialDate.leapYearCount(y - 1);
434         int mm = SerialDate.AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[m];
435         if (m > MonthConstants.FEBRUARY) {
436             if (SerialDate.isLeapYear(y)) {
437                 mm = mm + 1;
438             }
439         }
440         final int dd = d;
```

```
441         return yy + mm + dd + 1;
442     }
443
444     /**
445      * Calculate the day, month and year from the serial number.
446      */
447     private void calcDayMonthYear() {
448
449         // get the year from the serial date
450         final int days = this.serial - SERIAL_LOWER_BOUND;
451         // overestimated because we ignored leap days
452         final int overestimatedYYYY = 1900 + (days / 365);
453         final int leaps = SerialDate.leapYearCount(overestimatedYYYY);
454         final int nonleapdays = days - leaps;
455         // underestimated because we overestimated years
456         int underestimatedYYYY = 1900 + (nonleapdays / 365);
457
458         if (underestimatedYYYY == overestimatedYYYY) {
459             this.year = underestimatedYYYY;
460         }
461         else {
462             int ss1 = calcSerial(1, 1, underestimatedYYYY);
463             while (ss1 <= this.serial) {
464                 underestimatedYYYY = underestimatedYYYY + 1;
465                 ss1 = calcSerial(1, 1, underestimatedYYYY);
466             }
467             this.year = underestimatedYYYY - 1;
468         }
469
470         final int ss2 = calcSerial(1, 1, this.year);
471
472         int[] daysToEndOfPrecedingMonth
473             = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
474
475         if (isLeapYear(this.year)) {
476             daysToEndOfPrecedingMonth
477                 = LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
478         }
479
480         // get the month from the serial date
481         int mm = 1;
482         int sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
483         while (sss < this.serial) {
484             mm = mm + 1;
485             sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
486         }
487         this.month = mm - 1;
488
489         // what's left is d(+1);
490         this.day = this.serial - ss2
491                    - daysToEndOfPrecedingMonth[this.month] + 1;
492
493     }
494
495 }
```

Listing B-6

RelativeDayOfWeekRule.java

```
 1 /* ===========================================================================
 2  * JCommon : a free general purpose class library for the Java(tm) platform
 3  * ===========================================================================
 4  *
 5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 6  *
 7  * Project Info:  http://www.jfree.org/jcommon/index.html
 8  *
 9  * This library is free software; you can redistribute it and/or modify it
10  * under the terms of the GNU Lesser General Public License as published by
11  * the Free Software Foundation; either version 2.1 of the License, or
12  * (at your option) any later version.
13  *
14  * This library is distributed in the hope that it will be useful, but
15  * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
16  * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
17  * License for more details.
18  *
19  * You should have received a copy of the GNU Lesser General Public
20  * License along with this library; if not, write to the Free Software
21  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
22  * USA.
23  *
24  * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
25  * in the United States and other countries.]
26  *
27  * --------------------------
28  * RelativeDayOfWeekRule.java
29  * --------------------------
30  * (C) Copyright 2000-2003, by Object Refinery Limited and Contributors.
31  *
32  * Original Author:  David Gilbert (for Object Refinery Limited);
33  * Contributor(s):   -;
34  *
35  * $Id: RelativeDayOfWeekRule.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
36  *
37  * Changes (from 26-Oct-2001)
38  * --------------------------
39  * 26-Oct-2001 : Changed package to com.jrefinery.date.*;
40  * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
41  *
42  */
43
44 package org.jfree.date;
45
46 /**
47  * An annual date rule that returns a date for each year based on (a) a
48  * reference rule; (b) a day of the week; and (c) a selection parameter
49  * (SerialDate.PRECEDING, SerialDate.NEAREST, SerialDate.FOLLOWING).
50  * <P>
51  * For example, Good Friday can be specified as 'the Friday PRECEDING Easter
52  * Sunday'.
53  *
54  * @author David Gilbert
55  */
56 public class RelativeDayOfWeekRule extends AnnualDateRule {
57
58     /** A reference to the annual date rule on which this rule is based. */
59     private AnnualDateRule subrule;
60
61     /**
62      * The day of the week (SerialDate.MONDAY, SerialDate.TUESDAY, and so on).
63      */
64     private int dayOfWeek;
```

```
65
66      /** Specifies which day of the week (PRECEDING, NEAREST or FOLLOWING). */
67      private int relative;
68
69      /**
70       * Default constructor - builds a rule for the Monday following 1 January.
71       */
72      public RelativeDayOfWeekRule() {
73          this(new DayAndMonthRule(), SerialDate.MONDAY, SerialDate.FOLLOWING);
74      }
75
76      /**
77       * Standard constructor - builds rule based on the supplied sub-rule.
78       *
79       * @param subrule  the rule that determines the reference date.
80       * @param dayOfWeek  the day-of-the-week relative to the reference date.
81       * @param relative  indicates *which* day-of-the-week (preceding, nearest
82       *                  or following).
83       */
84      public RelativeDayOfWeekRule(final AnnualDateRule subrule,
85              final int dayOfWeek, final int relative) {
86          this.subrule = subrule;
87          this.dayOfWeek = dayOfWeek;
88          this.relative = relative;
89      }
90
91      /**
92       * Returns the sub-rule (also called the reference rule).
93       *
94       * @return The annual date rule that determines the reference date for this
95       *         rule.
96       */
97      public AnnualDateRule getSubrule() {
98          return this.subrule;
99      }
100
101     /**
102      * Sets the sub-rule.
103      *
104      * @param subrule  the annual date rule that determines the reference date
105      *                 for this rule.
106      */
107     public void setSubrule(final AnnualDateRule subrule) {
108         this.subrule = subrule;
109     }
110
111     /**
112      * Returns the day-of-the-week for this rule.
113      *
114      * @return the day-of-the-week for this rule.
115      */
116     public int getDayOfWeek() {
117         return this.dayOfWeek;
118     }
119
120     /**
121      * Sets the day-of-the-week for this rule.
122      *
123      * @param dayOfWeek  the day-of-the-week (SerialDate.MONDAY,
124      *                   SerialDate.TUESDAY, and so on).
125      */
126     public void setDayOfWeek(final int dayOfWeek) {
127         this.dayOfWeek = dayOfWeek;
128     }
129
```

```
130     /**
131      * Returns the 'relative' attribute, that determines *which*
132      * day-of-the-week we are interested in (SerialDate.PRECEDING,
133      * SerialDate.NEAREST or SerialDate.FOLLOWING).
134      *
135      * @return The 'relative' attribute.
136      */
137     public int getRelative() {
138         return this.relative;
139     }
140
141     /**
142      * Sets the 'relative' attribute (SerialDate.PRECEDING, SerialDate.NEAREST,
143      * SerialDate.FOLLOWING).
144      *
145      * @param relative  determines *which* day-of-the-week is selected by this
146      *                  rule.
147      */
148     public void setRelative(final int relative) {
149         this.relative = relative;
150     }
151
152     /**
153      * Creates a clone of this rule.
154      *
155      * @return a clone of this rule.
156      *
157      * @throws CloneNotSupportedException this should never happen.
158      */
159     public Object clone() throws CloneNotSupportedException {
160         final RelativeDayOfWeekRule duplicate
161             = (RelativeDayOfWeekRule) super.clone();
162         duplicate.subrule = (AnnualDateRule) duplicate.getSubrule().clone();
163         return duplicate;
164     }
165
166     /**
167      * Returns the date generated by this rule, for the specified year.
168      *
169      * @param year  the year (1900 &amp;lt;= year &amp;lt;= 9999).
170      *
171      * @return The date generated by the rule for the given year (possibly
172      *         <code>null</code>).
173      */
174     public SerialDate getDate(final int year) {
175
176         // check argument...
177         if ((year < SerialDate.MINIMUM_YEAR_SUPPORTED)
178             || (year > SerialDate.MAXIMUM_YEAR_SUPPORTED)) {
179             throw new IllegalArgumentException(
180                 "RelativeDayOfWeekRule.getDate(): year outside valid range.");
181         }
182
183         // calculate the date...
184         SerialDate result = null;
185         final SerialDate base = this.subrule.getDate(year);
186
187         if (base != null) {
188             switch (this.relative) {
189                 case(SerialDate.PRECEDING):
190                     result = SerialDate.getPreviousDayOfWeek(this.dayOfWeek,
191                             base);
```

```
192                     break;
193                 case(SerialDate.NEAREST):
194                     result = SerialDate.getNearestDayOfWeek(this.dayOfWeek,
195                             base);
196                     break;
197                 case(SerialDate.FOLLOWING):
198                     result = SerialDate.getFollowingDayOfWeek(this.dayOfWeek,
199                             base);
200                     break;
201                 default:
202                     break;
203             }
204         }
205         return result;
206
207     }
208
209 }
```

Listing B-7

DayDate.java (Final)

```
 1 /* ===========================================================================
 2  * JCommon : a free general purpose class library for the Java(tm) platform
 3  * ===========================================================================
 4  *
 5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
...
36  */
37 package org.jfree.date;
38
39 import java.io.Serializable;
40 import java.util.*;
41
42 /**
43  * An abstract class that represents immutable dates with a precision of
44  * one day.  The implementation will map each date to an integer that
45  * represents an ordinal number of days from some fixed origin.
46  *
47  * Why not just use java.util.Date?  We will, when it makes sense.  At times,
48  * java.util.Date can be *too* precise - it represents an instant in time,
49  * accurate to 1/1000th of a second (with the date itself depending on the
50  * time-zone).  Sometimes we just want to represent a particular day (e.g. 21
51  * January 2015) without concerning ourselves about the time of day, or the
52  * time-zone, or anything else.  That's what we've defined DayDate for.
53  *
54  * Use DayDateFactory.makeDate to create an instance.
55  *
56  * @author David Gilbert
57  * @author Robert C. Martin did a lot of refactoring.
58  */
59
60 public abstract class DayDate implements Comparable, Serializable {
61   public abstract int getOrdinalDay();
62   public abstract int getYear();
63   public abstract Month getMonth();
64   public abstract int getDayOfMonth();
65
66   protected abstract Day getDayOfWeekForOrdinalZero();
67
68   public DayDate plusDays(int days) {
69     return DayDateFactory.makeDate(getOrdinalDay() + days);
70   }
71
72   public DayDate plusMonths(int months) {
73     int thisMonthAsOrdinal = getMonth().toInt() - Month.JANUARY.toInt();
74     int thisMonthAndYearAsOrdinal = 12 * getYear() + thisMonthAsOrdinal;
75     int resultMonthAndYearAsOrdinal = thisMonthAndYearAsOrdinal + months;
76     int resultYear = resultMonthAndYearAsOrdinal / 12;
77     int resultMonthAsOrdinal = resultMonthAndYearAsOrdinal % 12 + Month.JANUARY.toInt();
78     Month resultMonth = Month.fromInt(resultMonthAsOrdinal);
79     int resultDay = correctLastDayOfMonth(getDayOfMonth(), resultMonth, resultYear);
80     return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
81   }
82
83   public DayDate plusYears(int years) {
84     int resultYear = getYear() + years;
85     int resultDay = correctLastDayOfMonth(getDayOfMonth(), getMonth(), resultYear);
86     return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
87   }
88
89   private int correctLastDayOfMonth(int day, Month month, int year) {
90     int lastDayOfMonth = DateUtil.lastDayOfMonth(month, year);
91     if (day > lastDayOfMonth)
92         day = lastDayOfMonth;
93     return day;
94   }
```

```
 95
 96   public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
 97     int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
 98     if (offsetToTarget >= 0)
 99       offsetToTarget -= 7;
100     return plusDays(offsetToTarget);
101   }
102
103   public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
104     int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
105     if (offsetToTarget <= 0)
106       offsetToTarget += 7;
107     return plusDays(offsetToTarget);
108   }
109
110   public DayDate getNearestDayOfWeek(Day targetDayOfWeek) {
111     int offsetToThisWeeksTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
112     int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
113     int offsetToPreviousTarget = offsetToFutureTarget - 7;
114
115     if (offsetToFutureTarget > 3)
116       return plusDays(offsetToPreviousTarget);
117     else
118       return plusDays(offsetToFutureTarget);
119   }
120
121   public DayDate getEndOfMonth() {
122     Month month = getMonth();
123     int year = getYear();
124     int lastDay = DateUtil.lastDayOfMonth(month, year);
125     return DayDateFactory.makeDate(lastDay, month, year);
126   }
127
128   public Date toDate() {
129     final Calendar calendar = Calendar.getInstance();
130     int ordinalMonth = getMonth().toInt() - Month.JANUARY.toInt();
131     calendar.set(getYear(), ordinalMonth, getDayOfMonth(), 0, 0, 0);
132     return calendar.getTime();
133   }
134
135   public String toString() {
136     return String.format("%02d-%s-%d", getDayOfMonth(), getMonth(), getYear());
137   }
138
139   public Day getDayOfWeek() {
140     Day startingDay = getDayOfWeekForOrdinalZero();
141     int startingOffset = startingDay.toInt() - Day.SUNDAY.toInt();
142     int ordinalOfDayOfWeek = (getOrdinalDay() + startingOffset) % 7;
143     return Day.fromInt(ordinalOfDayOfWeek + Day.SUNDAY.toInt());
144   }
145
146   public int daysSince(DayDate date) {
147     return getOrdinalDay() - date.getOrdinalDay();
148   }
149
150   public boolean isOn(DayDate other) {
151     return getOrdinalDay() == other.getOrdinalDay();
152   }
153
154   public boolean isBefore(DayDate other) {
155     return getOrdinalDay() < other.getOrdinalDay();
156   }
157
158   public boolean isOnOrBefore(DayDate other) {
159     return getOrdinalDay() <= other.getOrdinalDay();
```

```
160   }
161
162   public boolean isAfter(DayDate other) {
163     return getOrdinalDay() > other.getOrdinalDay();
164   }
165
166   public boolean isOnOrAfter(DayDate other) {
167     return getOrdinalDay() >= other.getOrdinalDay();
168   }
169
170   public boolean isInRange(DayDate d1, DayDate d2) {
171     return isInRange(d1, d2, DateInterval.CLOSED);
172   }
173
174   public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
175     int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
176     int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
177     return interval.isIn(getOrdinalDay(), left, right);
178   }
179 }
```

Listing B-8

Month.java (Final)

```
 1 package org.jfree.date;
 2
 3 import java.text.DateFormatSymbols;
 4
 5 public enum Month {
 6   JANUARY(1), FEBRUARY(2), MARCH(3),
 7   APRIL(4),   MAY(5),      JUNE(6),
 8   JULY(7),    AUGUST(8),   SEPTEMBER(9),
 9   OCTOBER(10),NOVEMBER(11),DECEMBER(12);
10   private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
11   private static final int[] LAST_DAY_OF_MONTH =
12     {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
13
14   private int index;
15
16   Month(int index) {
17     this.index = index;
18   }
19
20   public static Month fromInt(int monthIndex) {
21     for (Month m : Month.values()) {
22       if (m.index == monthIndex)
23         return m;
24     }
25     throw new IllegalArgumentException("Invalid month index " + monthIndex);
26   }
27
28   public int lastDay() {
29     return LAST_DAY_OF_MONTH[index];
30   }
31
32   public int quarter() {
33     return 1 + (index - 1) / 3;
34   }
35
36   public String toString() {
37     return dateFormatSymbols.getMonths()[index - 1];
38   }
39
40   public String toShortString() {
41     return dateFormatSymbols.getShortMonths()[index - 1];
42   }
43
44   public static Month parse(String s) {
45     s = s.trim();
46     for (Month m : Month.values())
47       if (m.matches(s))
48         return m;
49
50     try {
51       return fromInt(Integer.parseInt(s));
52     }
53     catch (NumberFormatException e) {}
54     throw new IllegalArgumentException("Invalid month " + s);
55   }
56
57   private boolean matches(String s) {
58     return s.equalsIgnoreCase(toString()) ||
59            s.equalsIgnoreCase(toShortString());
60   }
61
62   public int toInt() {
63     return index;
64   }
65 }
```

Listing B-9

Day.java (Final)

```
 1 package org.jfree.date;
 2
 3 import java.util.Calendar;
 4 import java.text.DateFormatSymbols;
 5
 6 public enum Day {
 7   MONDAY(Calendar.MONDAY),
 8   TUESDAY(Calendar.TUESDAY),
 9   WEDNESDAY(Calendar.WEDNESDAY),
10   THURSDAY(Calendar.THURSDAY),
11   FRIDAY(Calendar.FRIDAY),
12   SATURDAY(Calendar.SATURDAY),
13   SUNDAY(Calendar.SUNDAY);
14
15   private final int index;
16   private static DateFormatSymbols dateSymbols = new DateFormatSymbols();
17
18   Day(int day) {
19     index = day;
20   }
21
22   public static Day fromInt(int index) throws IllegalArgumentException {
23     for (Day d : Day.values())
24       if (d.index == index)
25         return d;
26     throw new IllegalArgumentException(
27       String.format("Illegal day index: %d.", index));
28   }
29
30   public static Day parse(String s) throws IllegalArgumentException {
31     String[] shortWeekdayNames =
32       dateSymbols.getShortWeekdays();
33     String[] weekDayNames =
34       dateSymbols.getWeekdays();
35
36     s = s.trim();
37     for (Day day : Day.values()) {
38       if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
39           s.equalsIgnoreCase(weekDayNames[day.index])) {
40         return day;
41       }
42     }
43     throw new IllegalArgumentException(
44       String.format("%s is not a valid weekday string", s));
45   }
46
47   public String toString() {
48     return dateSymbols.getWeekdays()[index];
49   }
50
51   public int toInt() {
52     return index;
53   }
54 }
```

Listing B-10

DateInterval.java (Final)

```
1 package org.jfree.date;
2
3 public enum DateInterval {
4   OPEN {
5     public boolean isIn(int d, int left, int right) {
6       return d > left && d < right;
7     }
8   },
9   CLOSED_LEFT {
10    public boolean isIn(int d, int left, int right) {
11      return d >= left && d < right;
12    }
13  },
14  CLOSED_RIGHT {
15    public boolean isIn(int d, int left, int right) {
16      return d > left && d <= right;
17    }
18  },
19  CLOSED {
20    public boolean isIn(int d, int left, int right) {
21      return d >= left && d <= right;
22    }
23  };
24
25  public abstract boolean isIn(int d, int left, int right);
26 }
```

Listing B-11

WeekInMonth.java (Final)

```
1 package org.jfree.date;
2
3 public enum WeekInMonth {
4   FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
5   private final int index;
6
7   WeekInMonth(int index) {
8     this.index = index;
9   }
10
11  public int toInt() {
12    return index;
13  }
14 }
```

Listing B-12

WeekdayRange.java (Final)

```
1 package org.jfree.date;
2
3 public enum WeekdayRange {
4   LAST, NEAREST, NEXT
5 }
```

Listing B-13

DateUtil.java (Final)

```
 1 package org.jfree.date;
 2
 3 import java.text.DateFormatSymbols;
 4
 5 public class DateUtil {
 6   private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
 7
 8   public static String[] getMonthNames() {
 9     return dateFormatSymbols.getMonths();
10   }
11
12   public static boolean isLeapYear(int year) {
13     boolean fourth = year % 4 == 0;
14     boolean hundredth = year % 100 == 0;
15     boolean fourHundredth = year % 400 == 0;
16     return fourth && (!hundredth || fourHundredth);
17   }
18
19   public static int lastDayOfMonth(Month month, int year) {
20     if (month == Month.FEBRUARY && isLeapYear(year))
21       return month.lastDay() + 1;
22     else
23       return month.lastDay();
24   }
25
26   public static int leapYearCount(int year) {
27     int leap4 = (year - 1896) / 4;
28     int leap100 = (year - 1800) / 100;
29     int leap400 = (year - 1600) / 400;
30     return leap4 - leap100 + leap400;
31   }
32 }
```

Listing B-14

DayDateFactory.java (Final)

```
 1 package org.jfree.date;
 2
 3 public abstract class DayDateFactory {
 4   private static DayDateFactory factory = new SpreadsheetDateFactory();
 5   public static void setInstance(DayDateFactory factory) {
 6     DayDateFactory.factory = factory;
 7   }
 8
 9   protected abstract DayDate _makeDate(int ordinal);
10   protected abstract DayDate _makeDate(int day, Month month, int year);
11   protected abstract DayDate _makeDate(int day, int month, int year);
12   protected abstract DayDate _makeDate(java.util.Date date);
13   protected abstract int _getMinimumYear();
14   protected abstract int _getMaximumYear();
15
16   public static DayDate makeDate(int ordinal) {
17     return factory._makeDate(ordinal);
18   }
19
20   public static DayDate makeDate(int day, Month month, int year) {
21     return factory._makeDate(day, month, year);
22   }
23
24   public static DayDate makeDate(int day, int month, int year) {
25     return factory._makeDate(day, month, year);
26   }
27
28   public static DayDate makeDate(java.util.Date date) {
29     return factory._makeDate(date);
30   }
31
32   public static int getMinimumYear() {
33     return factory._getMinimumYear();
34   }
35
36   public static int getMaximumYear() {
37     return factory._getMaximumYear();
38   }
39 }
```

Listing B-15

SpreadsheetDateFactory.java (Final)

```
1 package org.jfree.date;
2
3 import java.util.*;
4
5 public class SpreadsheetDateFactory extends DayDateFactory {
6   public DayDate _makeDate(int ordinal) {
7     return new SpreadsheetDate(ordinal);
8   }
9
10  public DayDate _makeDate(int day, Month month, int year) {
11    return new SpreadsheetDate(day, month, year);
12  }
13
14  public DayDate _makeDate(int day, int month, int year) {
15    return new SpreadsheetDate(day, month, year);
16  }
17
18  public DayDate _makeDate(Date date) {
19    final GregorianCalendar calendar = new GregorianCalendar();
20    calendar.setTime(date);
21    return new SpreadsheetDate(
22      calendar.get(Calendar.DATE),
23      Month.fromInt(calendar.get(Calendar.MONTH) + 1),
24      calendar.get(Calendar.YEAR));
25  }
26
27  protected int _getMinimumYear() {
28    return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
29  }
30
31  protected int _getMaximumYear() {
32    return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
33  }
34 }
```

Listing B-16

`SpreadsheetDate.java (Final)`

```
  1 /* ===========================================================================
  2  * JCommon : a free general purpose class library for the Java(tm) platform
  3  * ===========================================================================
  4  *
  5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
  6  *
...
 52  *
 53  */
 54
 55 package org.jfree.date;
 56
 57 import static org.jfree.date.Month.FEBRUARY;
 58
 59 import java.util.*;
 60
 61 /**
 62  * Represents a date using an integer, in a similar fashion to the
 63  * implementation in Microsoft Excel.  The range of dates supported is
 64  * 1-Jan-1900 to 31-Dec-9999.
 65  * <p/>
 66  * Be aware that there is a deliberate bug in Excel that recognises the year
 67  * 1900 as a leap year when in fact it is not a leap year. You can find more
 68  * information on the Microsoft website in article Q181370:
 69  * <p/>
 70  * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
 71  * <p/>
 72  * Excel uses the convention that 1-Jan-1900 = 1.  This class uses the
 73  * convention 1-Jan-1900 = 2.
 74  * The result is that the day number in this class will be different to the
 75  * Excel figure for January and February 1900...but then Excel adds in an extra
 76  * day (29-Feb-1900 which does not actually exist!) and from that point forward
 77  * the day numbers will match.
 78  *
 79  * @author David Gilbert
 80  */
 81 public class SpreadsheetDate extends DayDate {
 82   public static final int EARLIEST_DATE_ORDINAL = 2;     // 1/1/1900
 83   public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
 84   public static final int MINIMUM_YEAR_SUPPORTED = 1900;
 85   public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
 86   static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
 87     {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
 88   static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
 89     {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
 90
 91   private int ordinalDay;
 92   private int day;
 93   private Month month;
 94   private int year;
 95
 96   public SpreadsheetDate(int day, Month month, int year) {
 97     if (year < MINIMUM_YEAR_SUPPORTED || year > MAXIMUM_YEAR_SUPPORTED)
 98       throw new IllegalArgumentException(
 99         "The 'year' argument must be in range " +
100         MINIMUM_YEAR_SUPPORTED + " to " + MAXIMUM_YEAR_SUPPORTED + ".");
101     if (day < 1 || day > DateUtil.lastDayOfMonth(month, year))
102       throw new IllegalArgumentException("Invalid 'day' argument.");
103
104     this.year = year;
105     this.month = month;
```

```
106     this.day = day;
107     ordinalDay = calcOrdinal(day, month, year);
108   }
109
110   public SpreadsheetDate(int day, int month, int year) {
111     this(day, Month.fromInt(month), year);
112   }
113
114   public SpreadsheetDate(int serial) {
115     if (serial < EARLIEST_DATE_ORDINAL || serial > LATEST_DATE_ORDINAL)
116       throw new IllegalArgumentException(
117         "SpreadsheetDate: Serial must be in range 2 to 2958465.");
118
119     ordinalDay = serial;
120     calcDayMonthYear();
121   }
122
123   public int getOrdinalDay() {
124     return ordinalDay;
125   }
126
127   public int getYear() {
128     return year;
129   }
130
131   public Month getMonth() {
132     return month;
133   }
134
135   public int getDayOfMonth() {
136     return day;
137   }
138
139   protected Day getDayOfWeekForOrdinalZero() {return Day.SATURDAY;}
140
141   public boolean equals(Object object) {
142     if (!(object instanceof DayDate))
143       return false;
144
145     DayDate date = (DayDate) object;
146     return date.getOrdinalDay() == getOrdinalDay();
147   }
148
149   public int hashCode() {
150     return getOrdinalDay();
151   }
152
153   public int compareTo(Object other) {
154     return daysSince((DayDate) other);
155   }
156
157   private int calcOrdinal(int day, Month month, int year) {
158     int leapDaysForYear = DateUtil.leapYearCount(year - 1);
159     int daysUpToYear = (year - MINIMUM_YEAR_SUPPORTED) * 365 + leapDaysForYear;
160     int daysUpToMonth = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[month.toInt()];
161     if (DateUtil.isLeapYear(year) && month.toInt() > FEBRUARY.toInt())
162       daysUpToMonth++;
163     int daysInMonth = day - 1;
164     return daysUpToYear + daysUpToMonth + daysInMonth + EARLIEST_DATE_ORDINAL;
165   }
166
```

```
167   private void calcDayMonthYear() {
168     int days = ordinalDay - EARLIEST_DATE_ORDINAL;
169     int overestimatedYear = MINIMUM_YEAR_SUPPORTED + days / 365;
170     int nonleapdays = days - DateUtil.leapYearCount(overestimatedYear);
171     int underestimatedYear = MINIMUM_YEAR_SUPPORTED + nonleapdays / 365;
172
173     year = huntForYearContaining(ordinalDay, underestimatedYear);
174     int firstOrdinalOfYear = firstOrdinalOfYear(year);
175     month = huntForMonthContaining(ordinalDay, firstOrdinalOfYear);
176     day = ordinalDay - firstOrdinalOfYear - daysBeforeThisMonth(month.toInt());
177   }
178
179   private Month huntForMonthContaining(int anOrdinal, int firstOrdinalOfYear) {
180     int daysIntoThisYear = anOrdinal - firstOrdinalOfYear;
181     int aMonth = 1;
182     while (daysBeforeThisMonth(aMonth) < daysIntoThisYear)
183       aMonth++;
184
185     return Month.fromInt(aMonth - 1);
186   }
187
188   private int daysBeforeThisMonth(int aMonth) {
189     if (DateUtil.isLeapYear(year))
190       return LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
191     else
192       return AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
193   }
194
195   private int huntForYearContaining(int anOrdinalDay, int startingYear) {
196     int aYear = startingYear;
197     while (firstOrdinalOfYear(aYear) <= anOrdinalDay)
198       aYear++;
199
200     return aYear - 1;
201   }
202
203   private int firstOrdinalOfYear(int year) {
204     return calcOrdinal(1, Month.JANUARY, year);
205   }
206
207   public static DayDate createInstance(Date date) {
208     GregorianCalendar calendar = new GregorianCalendar();
209     calendar.setTime(date);
210     return new SpreadsheetDate(calendar.get(Calendar.DATE),
211                                Month.fromInt(calendar.get(Calendar.MONTH) + 1),
212                                calendar.get(Calendar.YEAR));
213
214   }
215 }
```