```
Loading vc-cvs...
```

```
package caltool.schedule;

import mvp.*;
import java.util.Calendar;
import java.text.*;

/****
 *
 * Class Date is the basic unit of calendar time keeping, consisting of a day
 * of the week, numeric date, month, and year.
 */

public class Date extends Model implements Comparable {

    /**
     * Construct an empty Date.
     */
    public Date() {
        day = null;
        number = 0;
        month = null;
        year = 0;
    }

    /**
     * Construct a date from the given string.  Set the valid field to false if
     * the given string does not parse as a valid date.  Note that the invalid
     * state representation is used instead of throwing an exception because
     * some users may want to delay the processing of invalid dates, and hence
     * may not be interested in handling an exception.
     *
     * Use java.text.SimpleDateFormat and java.util.Calendar to do the work.
     * This means that the first time the constructor is invoked, the static
     * format and jCalendar data fields are initialized to new SimpleDateFormat
     * and Calendar objects, resp.  These static values are used in all
     * subsequent Date constructions.
     */
    public Date(String dateString) {

        constructJCalendarIfNecessary();

        try {
            jCalendar.setTime(format.parse(dateString));
            day = convertJavaDay(jCalendar.get(Calendar.DAY_OF_WEEK));
            number = jCalendar.get(Calendar.DAY_OF_MONTH);
            month = MonthName.values()[jCalendar.get(Calendar.MONTH)];
            year = jCalendar.get(Calendar.YEAR);
            jDate = jCalendar.getTime();
            valid = true;
        }
        catch (ParseException e) {
            valid = false;
        }

    }

    /**
     * Construct a date from the given field values.  See the additional
     * comments in the String-valued constructor.
     */
    public Date(DayName day, int number, MonthName month, int year) {

        constructJCalendarIfNecessary();

        this.day = day;
        this.number = number;
        this.month = month;
        this.year = year;

        if (valid =
                (day != null) &&
                (month != null) &&
                (((month == MonthName.January) ||
                  (month == MonthName.March) ||
                  (month == MonthName.May) ||
                  (month == MonthName.July) ||
                  (month == MonthName.August) ||
                  (month == MonthName.October) ||
                  (month == MonthName.December)) ? (number <= 31) :
                      ((month == MonthName.April) ||
                       (month == MonthName.June) ||
                       (month == MonthName.September) ||
                       (month == MonthName.November)) ? (number <= 30) :
                           ((year % 4 == 0) && (year % 400 != 0)) ?
                               (number <= 29) : (number <= 28)) &&
                ((year >= 1) && (year <= 9999))) {
            jCalendar.set(year - 1900, month.ordinal(), number);
            jDate = jCalendar.getTime();
        }
    }

    /**
     * Construct the static java.util.format and Calendar if this is the first
     * time the constructor has been called.
     */
    protected void constructJCalendarIfNecessary() {
        if (format == null) {
            format = (SimpleDateFormat)
                DateFormat.getDateInstance(DateFormat.MEDIUM);
            jCalendar = format.getCalendar();
        }

    }

    /**
     * Convert a java.util.Calendar.DAY_OF_WEEK number to a
     * caltool.schedule.DayName enum.  This is necessary because JFC does not
     * (necessarily) map the pseudo-enum day name literals to any particular
     * numeric sequence.  The last time I checked, Calendar.MONDAY == 2.
     */
    protected DayName convertJavaDay(int javaDayNum) {
```

```
112         switch (javaDayNum) {
113             case Calendar.SUNDAY: return DayName.Sunday;
114             case Calendar.MONDAY: return DayName.Monday;
115             case Calendar.TUESDAY: return DayName.Tuesday;
116             case Calendar.WEDNESDAY: return DayName.Wednesday;
117             case Calendar.THURSDAY: return DayName.Thursday;
118             case Calendar.FRIDAY: return DayName.Friday;
119             case Calendar.SATURDAY: return DayName.Saturday;
120             default: return null;       // To placate javac
121         }
122     }
123
124     /**
125      * Return true if this is a valid date.
126      */
127     public boolean isValid() {
128         return valid;
129     }
130
131     /**
132      * Return true if this is an empty date, indicated by the date number = 0.
133      */
134     public boolean isEmpty() {
135         return number == 0;
136     }
137
138     /**
139      * Return the string representation of this.
140      */
141     public String toString() {
142         return day.toString().concat(" ").concat(Integer.toString(number)).
143             concat(" ").concat(month.toString()).concat(" ").
144                 concat(Integer.toString(year));
145     }
146
147     /**
148      * Define equality for this as componentwise equality.
149      */
150     public boolean equals(Object obj) {
151         Date otherDate = (Date) obj;
152
153         return
154             day.equals(otherDate.day) &&
155             number == otherDate.number &&
156             month.equals(otherDate.month) &&
157             year == otherDate.year;
158     }
159
160     /**
161      * Define compareTo using java.util.Calendar.  The comparison of invalid
162      * dates is defined as follows: (1) invalid < valid; (2) invalid ==
163      * invalid.
164      *
165      */
166     public int compareTo(Object o) {
167         Date otherDate = (Date) o;
168
169         if ((! valid) && (! otherDate.valid)) {
170             return 0;
171         }
172         if ((! valid) && (otherDate.valid)) {
173             return -1;
174         }
175         if ((valid) && (! otherDate.valid)) {
176             return 1;
177         }
178
179         /*
180          * If both dates are valid, compare using java.util.Date.compareTo.
181          */
182         return jDate.compareTo(otherDate.jDate);
183     }
184
185     /**
186      * Define the hash code for this as the sum of the components.  This hash
187      * code is used in turn by ItemKey.hashCode.
188      */
189     public int hashCode() {
190         return day.hashCode() + number + month.hashCode() + year;
191     }
192
193     /*-*
194      * Derived data fields
195      */
196
197     /** One of the seven standard days of the week */
198     protected DayName day;
199
200     /** Numeric date in a month, between 1 and 31 */
201     protected int number;
202
203     /** One of the twelve months of the year */
204     protected MonthName month;
205
206     /** The four-digit year number. (Yes, this Calendar Tool has a Y10K
207         problem.)
208      */
209     protected int year;
210
211     /** True if this is a valud date */
212     protected boolean valid;
213
214     /** The JFC SimpleDateFormat object to use for date calculations. */
215     SimpleDateFormat format;
216
217     /** The JFC  object to use for date calculations. */
218     Calendar jCalendar;
219
220     /** The java.util.Date value that represents this' date.  In future, this
221         may be the only data rep of this, but for now we keep our own model
222         data fields around as well.  At present, the significant use of this
223         date rep is in this.compareTo. */
```

```
224      protected java.util.Date jDate;
225
226  }
```