```
// joi/8/terminal/Terminal.java
// (and terminal/Terminal.java)
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.io.*;

/**
 * Terminal provides a user-friendly interface to the standard System
 * input and output streams (in, out, and err).
 * <p>
 * A Terminal is an object.  In general, one is expected to instantiate
 * just one Terminal.  Although one might instantiate several, all will
 * share the same System streams.
 * <p>
 * A Terminal may either explicitly echo input, or not.  Echoing input
 * is useful, for example, when testing with I/O redirection.
 * <p>
 * Inspired by Cay Horstmann's Console Class.
 */

public class Terminal
{
    private boolean echo = false;
    private static BufferedReader in =
        new BufferedReader(new FileReader(FileDescriptor.in));


    // Print a prompt to the console without a newline.

    private void printPrompt( String prompt )
    {
        print( prompt );
        System.out.flush();
    }

    /**
     * Construct a Terminal that doesn't echo input.
     */

    public Terminal()
    {
        this( false );
    }

    /**
     * Construct a Terminal.
     *
     * @param echo whether or not input should be echoed.
     */

    public Terminal( boolean echo )
    {
        this.echo = echo;
    }

```



```
    /**
     * Read a line (terminated by a newline) from the Terminal.
     * @param prompt output string to prompt for input.
     *
     * @return the string (without the newline character),
     *  null if eof.
     */

    public String readLine( String prompt )
    {
        printPrompt(prompt);
        try {
            String line = in.readLine();
            if (echo) {
                println(line);
            }
            return line;
        }
        catch (IOException e) {
            return null;
        }
    }

    /**
     * Read a line (terminated by a newline) from the Terminal.
     *
     * @return the string (without the newline character).
     */

    public String readLine()
    {
        return readLine( "" );
    }

    // Read a line from the Terminal.  An end of file,
    // indicated by a null, raises a runtime exception.
    // Used only internally.

    private String readNonNullLine()
    {
        return readNonNullLine( "" );
    }

    // Read a line from the Terminal.  An end of file,
    // indicated by a null, raises a runtime exception.
    // Used only internally.

    private String readNonNullLine( String prompt )
    {
        String line = readLine( prompt );
        if (line == null ) {
            throw new RuntimeException( "End of File encountered." );
        }
        return line;
    }

```

```
113      /**
114       * Read a word from the Terminal.
115       * If an empty line is entered, try again.
116       * Words are terminated by whitespace.
117       * Leading whitespace is trimmed; the rest of the line
118       * is disposed of.
119       *
120       * @param prompt output string to prompt for input.
121       *
122       * @return the word read.
123       */
124
125      public String readWord( String prompt )
126      {
127          String line = readNonNullLine( prompt );
128          if (line.length() == 0) {
129              println( "Empty line.  Please try again." );
130              return readWord("");
131          }
132          line = line.trim();
133          for ( int i = 0; i < line.length(); i++ ) {
134              if ( Character.isWhitespace( line.charAt(i) ) ) {
135                  return line.substring( 0, i );
136              }
137          }
138          return line;
139      }
140
141      /**
142       * Read a word from the Terminal.
143       * If an empty line is entered, try again.
144       * Words are terminated by whitespace.
145       * Leading whitespace is trimmed; the rest of the line
146       * is disposed of.
147       *
148       * @return the word read.
149       */
150
151      public String readWord()
152      {
153          return readWord( "" );
154      }
155
156      /**
157       * Read a word from the Terminal.
158       * If an empty line is entered, throw an exception.
159       * Words are terminated by whitespace.
160       * Leading whitespace is trimmed; the rest of the line
161       * is disposed of.
162       *
163       * @param prompt output string to prompt for input.
164       * @return the word read.
165       *
166       * @throws RuntimeException if it reads an empty line.
167       */
168
```



```
169      public String readWordOnce( String prompt )
170      {
171          String line = readNonNullLine( prompt );
172          if (line.length() == 0) {
173              throw new RuntimeException("Empty line encountered.");
174          }
175          line = line.trim();
176          for ( int i = 0; i < line.length(); i++ ) {
177              if ( Character.isWhitespace( line.charAt(i) ) ) {
178                  return line.substring( 0, i );
179              }
180          }
181          return line;
182      }
183
184      /**
185       * Read a word from the Terminal.
186       * If an empty line is entered, throw an exception.
187       * Words are terminated by whitespace.
188       * Leading whitespace is trimmed; the rest of the line
189       * is disposed of.
190       *
191       * @return the word read.
192       *
193       * @throws RuntimeException if it reads an empty line.
194       */
195
196      public String readWordOnce()
197      {
198          return readWordOnce( "" );
199      }
200
201      /**
202       * Read a character from the Terminal.
203       * Prompt again when an empty line is read.
204       *
205       * @param prompt output string to prompt for input.
206       *
207       * @return the character read.
208       */
209
210      public char readChar( String prompt )
211      {
212          String line = readNonNullLine(prompt);
213          if (line.length() == 0) {
214              println( "No character on line.  Please try again." );
215              return readChar("");
216          }
217          return line.charAt(0);
218      }
219
220      /**
221       * Read a character from the Terminal.
222       * Throw an exception if an empty line is read.
223       *
224       * @param prompt output string to prompt for input.
```

```
225       * @return the character read.
226       *
227       * @throws RuntimeException if it reads an empty line.
228       */
229
230      public char readCharOnce( String prompt )
231      {
232          String line = readNonNullLine(prompt);
233          if (line.length() == 0) {
234              throw new RuntimeException("Empty line encountered.");
235          }
236          return line.charAt(0);
237      }
238
239      /**
240       * Read a character from the Terminal.
241       * Prompt again when an empty line is read.
242       *
243       * @param prompt output string to prompt for input.
244       *
245       * @return the character read.
246       */
247
248      public char readChar()
249      {
250          return readChar("");
251      }
252
253      /**
254       * Read a character from the Terminal.
255       * Throw an exception if an empty line is read.
256       *
257       * @return the character read.
258       *
259       * @throws RuntimeException if it reads an empty line.
260       */
261
262      public char readCharOnce()
263      {
264          return readCharOnce("");
265      }
266
267      /**
268       * Read "yes" or "no" from the Terminal.
269       * If an empty line or improper character is read,
270       * try again.
271       * Look only at first character and accept any case.
272       *
273       * @param prompt output string to prompt for input.
274       * @return true if yes, false if no.
275       */
276
277      public boolean readYesOrNo( String prompt )
278      {
279          printPrompt( prompt );
280          while ( true ) {
```



```
281              char answer = readChar( " (y or n): ");
282              if ( answer == 'y' || answer == 'Y' ) {
283                  return true;
284              }
285              else if ( answer == 'n' || answer == 'N' ) {
286                  return false;
287              }
288              else {
289                  printPrompt( "oops!" );
290              }
291          }
292      }
293
294      /**
295       * Read "yes" or "no" from the Terminal.
296       * If an empty line or improper character is read,
297       * throw an exception.
298       * Look only at first character and accept any case.
299       *
300       * @param prompt output string to prompt for input.
301       * @return true if yes, false if no.
302       *
303       * @throws RuntimeException on improper input.
304       */
305
306      public boolean readYesOrNoOnce( String prompt )
307      {
308          printPrompt( prompt );
309          while ( true ) {
310              char answer = readCharOnce( " (y or n): ");
311              if ( answer == 'y' || answer == 'Y' ) {
312                  return true;
313              }
314              else if ( answer == 'n' || answer == 'N' ) {
315                  return false;
316              }
317              else {
318                  throw new RuntimeException( "Must be y or n." );
319              }
320          }
321      }
322
323      /**
324       * Read "yes" or "no" from the Terminal.
325       * If an empty line or improper character is read,
326       * try again. No prompting is done.
327       * Look only at first character and accept any case.
328       *
329       * @return true if yes, false if no.
330       */
331
332      public boolean readYesOrNo()
333      {
334          while ( true ) {
335              char answer = readChar();
336              if ( answer == 'y' || answer == 'Y' ) {
```

```
337                 return true;
338             }
339             else if ( answer == 'n' || answer == 'N' ) {
340                 return false;
341             }
342         }
343     }
344
345     /**
346      * Read "yes" or "no" from the Terminal.
347      * If an empty line or improper character is read,
348      * throw an exception.
349      * Look only at first character and accept any case.
350      *
351      * @return true if yes, false if no.
352      *
353      * @throws RuntimeException on improper input.
354      */
355
356     public boolean readYesOrNoOnce()
357     {
358         char answer = readCharOnce( " (y or n): ");
359         if ( answer == 'y' || answer == 'Y' ) {
360             return true;
361         }
362         else if ( answer == 'n' || answer == 'N' ) {
363             return false;
364         }
365         else {
366             throw new RuntimeException( "Must be y or n." );
367         }
368     }
369
370     /**
371      * Read an integer, terminated by a new line, from the Terminal.
372      * If a NumberFormatException is encountered, try again.
373      *
374      * @param prompt output string to prompt for input.
375      * @return the input value as an int.
376      */
377
378     public int readInt( String prompt )
379     {
380         while( true ) {
381             try {
382                 return Integer.
383                     parseInt(readNonNullLine( prompt ).trim());
384             }
385             catch (NumberFormatException e) {
386                 println( "Not an integer.  Please try again." );
387             }
388         }
389     }
390
391     /**
392      * Read an integer, terminated by a new line, from the Terminal.
```



```
393      *
394      * @param prompt output string to prompt for input.
395      * @return the input value as an int.
396      *
397      * @throws NumberFormatException for a badly formed integer.
398      */
399
400     public int readIntOnce( String prompt )
401         throws NumberFormatException
402     {
403         return Integer.parseInt(readNonNullLine( prompt ).trim());
404     }
405
406     /**
407      * Read an integer, terminated by a new line, from the Terminal.
408      * If a NumberFormatException is encountered, try again.
409      *
410      * @return the input value as an int.
411      */
412
413     public int readInt()
414     {
415         return readInt("");
416     }
417
418     /**
419      * Read an integer, terminated by a new line, from the Terminal.
420      *
421      * @return the input value as an int.
422      *
423      * @throws NumberFormatException for a badly formed integer.
424      */
425
426     public int readIntOnce()
427         throws NumberFormatException
428     {
429         return readIntOnce("");
430     }
431
432     /**
433      * Read a double-precision floating point number,
434      * terminated by a newline, from the Terminal.
435      * If a NumberFormatException is encountered, try again.
436      *
437      * @param prompt output string to prompt for input.
438      * @return the input value as a double.
439      */
440
441     public double readDouble( String prompt )
442     {
443         while( true ) {
444             try {
445                 return Double.
446                     parseDouble(readNonNullLine( prompt ).trim());
447             }
448             catch (NumberFormatException e) {
```

```
println("Not a floating point number. Please try again.");
            }
        }
    }

    /**
     * Read a double-precision floating point number,
     * terminated by a newline, from the Terminal.
     *
     * @param prompt output string to prompt for input.
     * @return the input value as a double.
     *
     * @throws NumberFormatException for a badly formed number.
     */

    public double readDoubleOnce( String prompt )
        throws NumberFormatException
    {
        return Double.parseDouble(readNonNullLine( prompt ).trim());
    }

    /**
     * Read a double-precision floating point number,
     * terminated by a newline, from the Terminal.
     * If a NumberFormatException is encountered, try again.
     *
     * @return the input value as a double.
     */

    public double readDouble()
    {
        return readDouble("");
    }

    /**
     * Read a double-precision floating point number,
     * terminated by a newline, from the Terminal.
     *
     * @return the input value as a double.
     *
     * @throws NumberFormatException for a badly formed number.
     */

    public double readDoubleOnce()
        throws NumberFormatException
    {
        return readDouble("");
    }

    /**
     * Print a Boolean value
     * (<code>true</code> or <code>false</code>)
     * to standard output (without a newline).
     *
     * @param b Boolean to print.
     */
```



```

    public void print( boolean b )
    {
        System.out.print( b );
    }

    /**
     * Print character to standard output (without a newline).
     *
     * @param ch character to print.
     */

    public void print( char ch )
    {
        System.out.print( ch );
    }

    /**
     * Print character array to standard output (without a newline).
     *
     * @param s character array to print.
     */

    public void print( char[] s )
    {
        System.out.print( s );
    }

    /**
     * Print a double-precision floating point number to standard
     * output (without a newline).
     *
     * @param val number to print.
     */

    public void print( double val )
    {
        System.out.print( val );
    }

    /**
     * Print a floating point number to standard output
     * (without a newline).
     *
     * @param val number to print.
     */

    public void print( float val )
    {
        System.out.print( val );
    }

    /**
     * Print integer to standard output (without a newline).
     *
     * @param val integer to print.
```

```
561       */
562
563      public void print( int val )
564      {
565          System.out.print( val );
566      }
567
568      /**
569       * Print a long integer to standard output (without a newline).
570       *
571       * @param val integer to print.
572       */
573
574      public void print( long val )
575      {
576          System.out.print( val );
577      }
578
579      /**
580       * Print Object to standard output (without a newline).
581       *
582       * @param val Object to print.
583       */
584
585      public void print( Object val )
586      {
587          System.out.print( val.toString() );
588      }
589
590      /**
591       * Print string to standard output (without a newline).
592       *
593       * @param str String to print.
594       */
595
596      public void print( String str )
597      {
598          System.out.print( str );
599      }
600
601      /**
602       * Print a newline to standard output,
603       * terminating the current line.
604       */
605
606      public void println()
607      {
608          System.out.println();
609      }
610
611      /**
612       * Print a Boolean value
613       * (<code>true</code> or <code>false</code>)
614       * to standard output, followed by a newline.
615       * @param b Boolean to print.
616       */
```



```
617
618      public void println( boolean b )
619      {
620          System.out.println( b );
621      }
622
623      /**
624       * Print character to standard output, followed by a newline.
625       *
626       * @param ch character to print.
627       */
628
629      public void println( char ch )
630      {
631          System.out.println( ch );
632      }
633
634      /**
635       * Print a character array to standard output,
636       * followed by a newline.
637       *
638       * @param s character array to print.
639       */
640
641      public void println( char[] s )
642      {
643          System.out.println( s );
644      }
645
646      /**
647       * Print floating point number to standard output,
648       * followed by a newline.
649       *
650       * @param val number to print.
651       */
652
653      public void println( float val )
654      {
655          System.out.println( val );
656      }
657
658      /**
659       * Print a double-precision floating point number to standard
660       * output, followed by a newline.
661       *
662       * @param val number to print.
663       */
664
665      public void println( double val )
666      {
667          System.out.println( val );
668      }
669
670      /**
671       * Print integer to standard output, followed by a newline.
672       *
```

```
673       * @param val integer to print.
674       */
675
676      public void println( int val )
677      {
678          System.out.println( val );
679      }
680
681      /**
682       * Print a long integer to standard output,
683       * followed by a newline.
684       *
685       * @param val long integer to print.
686       */
687
688      public void println( long val )
689      {
690          System.out.println( val );
691      }
692
693      /**
694       * Print Object to standard output, followed by a newline.
695       *
696       * @param val Object to print
697       */
698
699      public void println( Object val )
700      {
701          System.out.println( val.toString() );
702      }
703
704      /**
705       * Print string to standard output, followed by a newline.
706       *
707       * @param str String to print
708       */
709
710      public void println( String str )
711      {
712          System.out.println( str );
713      }
714
715      /**
716       * Print a Boolean value
717       * (<code>true</code> or <code>false</code>)
718       * to standard err (without a newline).
719       *
720       * @param b Boolean to print.
721       */
722
723      public void errPrint( boolean b )
724      {
725          System.err.print( b );
726      }
727
728      /**
```



```
729       * Print character to standard err (without a newline).
730       *
731       * @param ch character to print.
732       */
733
734      public void errPrint( char ch )
735      {
736          System.err.print( ch );
737      }
738
739      /**
740       * Print character array to standard err (without a newline).
741       *
742       * @param s character array to print.
743       */
744
745      public void errPrint( char[] s )
746      {
747          System.err.print( s );
748      }
749
750      /**
751       * Print a double-precision floating point number to standard
752       * err (without a newline).
753       *
754       * @param val number to print.
755       */
756
757      public void errPrint( double val )
758      {
759          System.err.print( val );
760      }
761
762      /**
763       * Print a floating point number to standard err
764       * (without a newline).
765       *
766       * @param val number to print.
767       */
768
769      public void errPrint( float val )
770      {
771          System.err.print( val );
772      }
773
774      /**
775       * Print integer to standard err (without a newline).
776       *
777       * @param val integer to print.
778       */
779
780      public void errPrint( int val )
781      {
782          System.err.print( val );
783      }
784
```

```
    /**
     * Print a long integer to standard err (without a newline).
     *
     * @param val integer to print.
     */

    public void errPrint( long val )
    {
        System.err.print( val );
    }

    /**
     * Print Object to standard err (without a newline).
     *
     * @param val Object to print.
     */

    public void errPrint( Object val )
    {
        System.err.print( val.toString() );
    }

    /**
     * Print string to standard err (without a newline).
     *
     * @param str String to print.
     */

    public void errPrint( String str )
    {
        System.err.print( str );
    }

    /**
     * Print a newline to standard err,
     * terminating the current line.
     */

    public void errPrintln()
    {
        System.err.println();
    }

    /**
     * Print a Boolean value
     * (<code>true</code> or <code>false</code>)
     * to standard err, followed by a newline.
     *
     * @param b Boolean to print.
     */

    public void errPrintln( boolean b )
    {
        System.err.println( b );
    }

```



```
    /**
     * Print character to standard err, followed by a newline.
     *
     * @param ch character to print.
     */

    public void errPrintln( char ch )
    {
        System.err.println( ch );
    }

    /**
     * Print a character array to standard err,
     * followed by a newline.
     *
     * @param s character array to print.
     */

    public void errPrintln( char[] s )
    {
        System.err.println( s );
    }

    /**
     * Print floating point number to standard err,
     * followed by a newline.
     *
     * @param val number to print.
     */

    public void errPrintln( float val )
    {
        System.err.println( val );
    }

    /**
     * Print a double-precision floating point number to
     * standard err, followed by a newline.
     *
     * @param val number to print.
     */

    public void errPrintln( double val )
    {
        System.err.println( val );
    }

    /**
     * Print integer to standard err, followed by a newline.
     *
     * @param val integer to print.
     */

    public void errPrintln( int val )
    {
        System.err.println( val );
```

```
897      }
898
899      /**
900       * Print a long integer to standard err, followed by a newline.
901       *
902       * @param val long integer to print.
903       */
904
905      public void errPrintln( long val )
906      {
907          System.err.println( val );
908      }
909
910      /**
911       * Print Object to standard err, followed by a newline.
912       *
913       * @param val Object to print
914       */
915
916      public void errPrintln( Object val )
917      {
918          System.err.println( val.toString() );
919      }
920
921      /**
922       * Print string to standard err, followed by a newline.
923       *
924       * @param str String to print
925       */
926
927      public void errPrintln( String str )
928      {
929          System.err.println( str );
930      }
931
932      /**
933       * Unit test for Terminal.
934       *
935       * @param args command line arguments:
936       * <pre>
937       *    -e echo all input.
938       * </pre>
939       */
940
941      public static void main( String[] args )
942      {
943          Terminal t =
944              new Terminal( args.length == 1 && args[0].equals("-e") );
945
946          String line = t.readLine( "line:" );
947          String word = t.readWord( "word:" );
948          char   c    = t.readChar( "char:" );
949          boolean yn  = t.readYesOrNo( "yorn:" );
950          double d    = t.readDouble( "double:" );
951          int    i    = t.readInt( "int:" );
952
```



```
953          t.print("  line:[");  t.print(line);    t.print("]");
954          t.print("  line:[");  t.println(line);  t.print("]");
955
956          t.print("  word:[");  t.print(word);    t.print("]");
957          t.print("  word:[");  t.println(word);  t.print("]");
958
959          t.print("  char:[");  t.print(c);       t.print("]");
960          t.print("  char:[");  t.println(c);     t.print("]");
961
962          t.print("  yorn:[");  t.print(yn);      t.print("]");
963          t.print("  yorn:[");  t.println(yn);    t.print("]");
964
965          t.print("  doub:[");  t.print(d);       t.print("]");
966          t.print("  doub:[");  t.println(d);     t.print("]");
967
968          t.print("  int:[");   t.print(i);       t.print("]");
969          t.print("  int:[");   t.println(i);     t.print("]");
970
971          t.errPrint("  line:[");  t.errPrint(line);    t.errPrint("]");
972          t.errPrint("  line:[");  t.errPrintln(line);  t.errPrint("]");
973
974          t.errPrint("  word:[");  t.errPrint(word);    t.errPrint("]");
975          t.errPrint("  word:[");  t.errPrintln(word);  t.errPrint("]");
976
977          t.errPrint("  char:[");  t.errPrint(c);       t.errPrint("]");
978          t.errPrint("  char:[");  t.errPrintln(c);     t.errPrint("]");
979
980          t.errPrint("  yorn:[");  t.errPrint(yn);      t.errPrint("]");
981          t.errPrint("  yorn:[");  t.errPrintln(yn);    t.errPrint("]");
982
983          t.errPrint("  doub:[");  t.errPrint(d);       t.errPrint("]");
984          t.errPrint("  doub:[");  t.errPrintln(d);     t.errPrint("]");
985
986          t.errPrint("  int:[");   t.errPrint(i);       t.errPrint("]");
987          t.errPrint("  int:[");   t.errPrintln(i);     t.errPrint("]");
988      }
989  }
```

```
// joi/8/juno/Password.java//
//
//  Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * Model a good password.
 *
 * <p>
 * A password is a String satisfying the following conditions
 * (close to those required of Unix passwords, according to
 * the <code> man passwd </code> command in Unix):
 * <br>
 * <ul>
 * <li> A password must have at least PASSLENGTH characters, where
 *      PASSLENGTH defaults to 6. Only the first eight characters
 *      are significant.
 *
 * <li> A password must contain at least two alphabetic characters
 *      and at least one numeric or special character. In this  case,
 *      "alphabetic" refers to all upper or lower case letters.
 *
 * <li> A password must not contain a specified string as a substring
 *      For comparison purposes, an upper case letter and its
 *      corresponding lower case letter are equivalent.
 *
 * <li> A password must not be a substring of a specified string.
 *      For comparison purposes, an upper case letter and its
 *      corresponding lower case letter are equivalent.
 *
 * </ul>
 * <br>
 * A password string may be stored in a Password object only in
 * encrypted form.
 */

public class Password
{
    private String password;

    /**
     * Construct a new Password.
     *
     * @param password the new password.
     * @param notSubstringOf a String that may not contain the password.
     * @param doesNotContain a String the password may not contain.
     *
     * @exception BadPasswordException when password is unacceptable.
     */

    public Password(String password, String notSubstringOf,
                    String doesNotContain)
         throws BadPasswordException
    {
        // if password is not acceptable
        // throw new BadPasswordException( reason )
```



```
        this.password = encrypt(password);
    }

    // Rewrite s in a form that makes it hard to guess s.

    private String encrypt( String s )
    {
        return Integer.toHexString(s.hashCode());
    }

    /**
     * See whether a supplied guess matches this password.
     *
     * @param guess the trial password.
     *
     * @exception BadPasswordException when match fails.
     */

    public void match(String guess)
         throws BadPasswordException
    {
    }

    /**
     * Unit test for Password objects.
     */

    public static void main( String[] args )
    {
    }
}
```

```
 1  // joi/8/juno/BadPasswordException.java
 2  //
 3  //
 4  //  Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  /**
 7   * The exception thrown when an initial password is unacceptable
 8   * or a match against an existing password fails.
 9   */
10
11  public class BadPasswordException extends Exception
12  {
13      BadPasswordException()
14      {
15          super();
16      }
17
18      BadPasswordException(String message)
19      {
20          super(message);
21      }
22  }
```