```
//  joi/6/juno/Juno.java
//
//
//  Copyright 2003 Bill Campbell and Ethan Bolker

import java.io.*;
import java.util.*;
import java.lang.*;

/**
 * Juno (Juno's Unix NOt) mimics a command line operating system
 * like Unix.
 * <p>
 * A Juno system has a name, a set of Users, a JFile system,
 * a login process and a set of shell commands.
 *
 * @see User
 * @see JFile
 * @see ShellCommand
 *
 * @version 6
 */

public class Juno
{
    private final static String os      = "Juno";
    private final static String version = "6";

    private String    hostName;   // host machine name
    private Map       users;      // lookup table for Users
    private Terminal  console;    // for input and output

    private Directory slash;      // root of JFile system
    private Directory userHomes;  // for home directories

    private ShellCommandTable commandTable; // shell commands

    /**
     * Construct a Juno (operating system) object.
     *
     * @param hostName  the name of the host on which it's running.
     * @param echoInput should all input be echoed as output?
     */

    public Juno( String hostName, boolean echoInput )
    {
        // initialize the Juno environment ...

        this.hostName = hostName;
        console       = new Terminal( echoInput );
        users         = new TreeMap();           // for registered Users
        commandTable  = new ShellCommandTable(); // for shell commands

        // the file system

        slash     = new Directory( "", null, null );
```



```
        User root = new User( "root", slash, "Rick Martin" );
        users.put( "root", root );
        slash.setOwner(root);
        userHomes = new Directory( "users", root, slash );

        // create, then start a command line login interpreter

        LoginInterpreter interpreter
            = new LoginInterpreter( this, console );
        interpreter.CLILogin();
    }

    /**
     * The name of the host computer on which this system
     * is running.
     *
     * @return the host computer name.
     */

    public String getHostName()
    {
        return hostName;
    }

    /**
     * The name of this operating system.
     *
     * @return the operating system name.
     */

    public String getOS()
    {
        return os;
    }

    /**
     * The version number for this system.
     *
     * @return the version number.
     */

    public String getVersion()
    {
        return version;
    }

    /**
     * The directory containing all user homes for this system.
     *
     * @return the directory containing user homes.
     */

    public Directory getUserHomes()
    {
        return userHomes;
    }
```

```
113
114     /**
115      * The shell command table for this system.
116      *
117      * @return the shell command table.
118      */
119
120     public ShellCommandTable getCommandTable()
121     {
122         return commandTable;
123     }
124
125     /**
126      * Look up a user by user name.
127      *
128      * @param username the user's name.
129      * @return the appropriate User object.
130      */
131
132     public User lookupUser( String username )
133     {
134         return (User) users.get( username );
135     }
136
137     /**
138      * Create a new User.
139      *
140      * @param userName the User's login name.
141      * @param home her home Directory.
142      * @param realName her real name.
143      * @return newly created User.
144      */
145
146     public User createUser( String userName, Directory home,
147                             String realName )
148     {
149         User newUser = new User( userName, home, realName );
150         users.put( userName, newUser );
151         return newUser;
152     }
153
154     /**
155      * The Juno system may be given the following command line
156      * arguments.
157      * <pre>
158      *
159      * -e:         Echo all input (useful for testing).
160      *
161      * -version:   Report the version number and exit.
162      *
163      * [hostname]: The name of the host on which
164      *             Juno is running (optional).
165      * </pre>
166      */
167
168     public static void main( String[] args )
```



```
169     {
170         // Parse command line options
171
172         boolean echoInput = false;
173         String  hostName  = "mars";
174
175         for (int i=0; i < args.length; i++) {
176             if (args[i].equals("-version")) {
177                 System.out.println( os + " version " + version );
178                 System.exit(0);
179             }
180             if (args[i].equals("-e")) {
181                 echoInput = true;
182             }
183             else {
184                 hostName = args[i];
185             }
186         }
187
188         // create a Juno instance, which will start itself
189
190         new Juno( hostName, echoInput );
191     }
192 }
```

```
// joi/6/juno/LoginInterpreter.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Interpreter for Juno login commands.
 *
 * There are so few commands that if-then-else logic is OK.
 *
 * @version 6
 */

public class LoginInterpreter
{
    private static final String LOGIN_COMMANDS =
        "help, register, <username>, exit";

    private Juno     system;  // the Juno object
    private Terminal console; // for i/o

    /**
     * Construct a new LoginInterpreter for interpreting
     * login commands.
     *
     * @param system the system creating this interpreter.
     * @param console the Terminal used for input and output.
     */

    public LoginInterpreter( Juno system, Terminal console)
    {
        this.system  = system;
        this.console = console;
    }

    /**
     * Set the console for this interpreter.  Used by the
     * creator of this interpreter.
     *
     * @param console the Terminal to be used for input and output.
     */

    public void setConsole( Terminal console )
    {
        this.console = console;
    }

    /**
     * Simulates behavior at login: prompt.
     * CLI stands for "Command Line Interface".
     */

    public void CLILogin()
    {
```



```
        welcome();
        boolean moreWork = true;
        while( moreWork ) {
            moreWork = interpret( console.readLine( "Juno login: " ) );
        }
    }

    // Parse user's command line and dispatch appropriate
    // semantic action.
    //
    // return true unless "exit" command or null inputLine.

    private boolean interpret( String inputLine )
    {
        if (inputLine == null) return false;
        StringTokenizer st =
            new StringTokenizer( inputLine );
        if (st.countTokens() == 0) {
            return true; // skip blank line
        }
        String visitor = st.nextToken();
        if (visitor.equals( "exit" )) {
            return false;
        }
        if (visitor.equals( "register" )) {
            register( st );
        }
        else if (visitor.equals( "help" )) {
            help();
        }
        else {
            User user = system.lookupUser(visitor);
            new Shell( system, user, console );
        }
        return true;
    }

    // Register a new user, giving him or her a login name and a
    // home directory on the system.
    //
    // StringTokenizer argument contains the new user's login name
    // followed by full real name.

    private void register( StringTokenizer st )
    {
        String userName = st.nextToken();
        String realName = st.nextToken("").trim();
        Directory home  = new Directory( userName, null,
                                         system.getUserHomes() );
        User user = system.createUser( userName, home, realName );
        home.setOwner( user );
    }

    // Display a short welcoming message, and remind users of
    // available commands.

```

```
   113      private void welcome()
   114      {
   115          console.println( "Welcome to " + system.getHostName() +
   116                           " running " + system.getOS() +
   117                           " version " + system.getVersion() );
   118          help();
   119      }
   120
   121      // Remind user of available commands.
   122
   123      private void help()
   124      {
   125          console.println( LOGIN_COMMANDS );
   126          console.println("");
   127      }
   128  }
```

```
 1  // joi/6/juno/Shell.java
 2  //
 3  //
 4  // Copyright 2003, Ethan Bolker and Bill Campbell
 5
 6  import java.util.*;
 7
 8  /**
 9   * Models a shell (command interpreter)
10   *
11   * The Shell knows the (Juno) system it's working in,
12   * the User who started it,
13   * and the console to which to send output.
14   *
15   * It keeps track of the the current working directory (.) .
16   *
17   * @version 6
18   */
19
20  public class Shell
21  {
22      private Juno system;       // the operating system object
23      private User user;         // the user logged in
24      private Terminal console;  // the console for this shell
25      private Directory dot;     // the current working directory
26
27      /**
28       * Construct a login shell for the given user and console.
29       *
30       * @param system a reference to the Juno system.
31       * @param user the User logging in.
32       * @param console a Terminal for input and output.
33       */
34
35      public Shell( Juno system, User user, Terminal console )
36      {
37          this.system  = system;
38          this.user    = user;
39          this.console = console;
40          dot          = user.getHome(); // default current directory
41          CLIShell(); // start the command line interpreter
42      }
43
44      // Run the command line interpreter
45
46      private void CLIShell()
47      {
48          boolean moreWork = true;
49          while(moreWork) {
50              moreWork = interpret( console.readLine( getPrompt() ) );
51          }
52          console.println("goodbye");
53      }
54
55      // Interpret a String of the form
56      //     shellcommand command-arguments
```



```
57      //
58      // return true, unless shell command is logout.
59
60      private boolean interpret( String inputLine )
61      {
62          StringTokenizer st = stripComments(inputLine);
63          if (st.countTokens() == 0) {
64              return true;             // skip blank line
65          }
66          String commandName = st.nextToken();
67          if (commandName.equals( "logout" )) {
68              return false;            // user is done
69          }
70          ShellCommand commandObject =
71              system.getCommandTable().lookup( commandName );
72          if (commandObject == null ) {
73              console.errPrintln( "Unknown command: " + commandName );
74          }
75          else {
76              commandObject.doIt( st, this );
77          }
78          return true;
79      }
80
81      // Strip characters from '#' to end of line, create and
82      // return a StringTokenizer for what's left.
83
84      private StringTokenizer stripComments( String line )
85      {
86          int commentIndex = line.indexOf('#');
87          if (commentIndex >= 0) {
88              line = line.substring(0,commentIndex);
89          }
90          return new StringTokenizer(line);
91      }
92
93      /**
94       * The prompt for the CLI.
95       *
96       * @return the prompt string.
97       */
98
99      public String getPrompt()
100     {
101         return system.getHostName() + "> ";
102     }
103
104     /**
105      * The User associated with this Shell.
106      *
107      * @return the user.
108      */
109
110     public User getUser()
111     {
112         return user;
```

```
113      }
114
115      /**
116       * The current working directory for this Shell.
117       *
118       * @return the current working directory.
119       */
120
121      public Directory getDot()
122      {
123          return dot;
124      }
125
126      /**
127       * Set the current working directory for this Shell.
128       *
129       * @param dot the new working directory.
130       */
131
132      public void setDot(Directory dot)
133      {
134          this.dot = dot;
135      }
136
137      /**
138       * The console associated with this Shell.
139       *
140       * @return the console.
141       */
142
143      public Terminal getConsole()
144      {
145          return console;
146      }
147
148      /**
149       * The Juno object associated with this Shell.
150       *
151       * @return the Juno instance that created this Shell.
152       */
153
154      public Juno getSystem()
155      {
156          return system;
157      }
158  }
```

```
 1  // joi/6/juno/ShellCommand.java
 2  //
 3  //
 4  // Copyright 2003 Ethan Bolker and Bill Campbell
 5
 6  import java.util.*;
 7
 8  /**
 9   * Model those features common to all ShellCommands.
10   *
11   * Each concrete extension of this class provides a constructor
12   * and an implementation for method doIt.
13   *
14   * @version 6
15   */
16
17  public abstract class ShellCommand
18  {
19      private String helpString;   // documents the command
20      private String argString;    // any args to the command
21
22      /**
23       * A constructor, always called (as super()) by the subclass.
24       * Used only for commands that have arguments.
25       *
26       * @param helpString a brief description of what the command does.
27       * @param argString a prototype illustrating the required arguments.
28       */
29
30      protected ShellCommand( String helpString, String argString )
31      {
32          this.argString  = argString;
33          this.helpString = helpString;
34      }
35
36      /**
37       * A constructor for commands having no arguments.
38       *
39       * @param helpString a brief description of what the command does.
40       */
41
42      protected ShellCommand( String helpString )
43      {
44          this( helpString, "" );
45      }
46
47      /**
48       * Execute the command.
49       *
50       * @param args the remainder of the command line.
51       * @param sh   the current shell
52       */
53
54      public abstract void doIt( StringTokenizer args, Shell sh );
55
56      /**
```



```
57       * Help for this command.
58       *
59       * @return the help string.
60       */
61
62      public String getHelpString()
63      {
64          return helpString;
65      }
66
67      /**
68       * The argument string prototype.
69       *
70       * @return the argument string prototype.
71       */
72
73      public String getArgString()
74      {
75          return argString;
76      }
77  }
```

```
 1  // joi/6/juno/MkdirCommand.java
 2  //
 3  //
 4  // Copyright 2003, Ethan Bolker and Bill Campbell
 5
 6  import java.util.*;
 7
 8  /**
 9   * The Juno shell command to create a new directory.
10   * Usage:
11   * <pre>
12   *     mkdir directory-name
13   * </pre>
14   *
15   * @version 6
16   */
17
18  public class MkdirCommand extends ShellCommand
19  {
20      /**
21       * Construct a MkdirCommand object.
22       */
23
24      public MkdirCommand()
25      {
26          super( "create a subdirectory of the current directory",
27                 "directory-name" );
28      }
29
30      /**
31       * Create a new Directory in the current Directory.
32       *
33       * @param args the remainder of the command line.
34       * @param sh the current shell
35       */
36
37      public void doIt( StringTokenizer args, Shell sh )
38      {
39          String filename = args.nextToken();
40          new Directory( filename, sh.getUser(), sh.getDot() );
41      }
42  }
```

```
 1  // joi/6/juno/TypeCommand.java
 2  //
 3  //
 4  // Copyright 2003, Ethan Bolker and Bill Campbell
 5
 6  import java.util.*;
 7
 8  /**
 9   * The Juno shell command to display the contents of a
10   * text file.
11   * Usage:
12   * <pre>
13   *     type textfile
14   * </pre>
15   *
16   * @version 6
17   */
18
19  public class TypeCommand extends ShellCommand
20  {
21      /**
22       * Construct a TypeCommand object.
23       */
24
25      TypeCommand()
26      {
27          super( "display contents of a TextFile", "textfile" );
28      }
29
30      /**
31       * Display the contents of a TextFile.
32       *
33       * @param args the reminder of the command line.
34       * @param sh the current Shell
35       */
36
37      public void doIt( StringTokenizer args, Shell sh )
38      {
39          String filename = args.nextToken();
40          sh.getConsole().println(
41              ( (TextFile) sh.getDot().
42                retrieveJFile( filename ) ).getContents() );
43      }
44  }
```

```
 1  // joi/6/juno/HelpCommand.java
 2  //
 3  //
 4  // Copyright 2003, Ethan Bolker and Bill Campbell
 5
 6  import java.util.*;
 7
 8  /**
 9   * The Juno shell command to display help on the shell commands.
10   * Usage:
11   * <pre>
12   *     help
13   * </pre>
14   *
15   * @version 6
16   */
17
18  public class HelpCommand extends ShellCommand
19  {
20      /**
21       * Construct a HelpCommand object.
22       */
23
24      HelpCommand()
25      {
26          super( "display ShellCommands" );
27      }
28
29      /**
30       * Display help for all commands.
31       *
32       * @param args the remainder of the command line.
33       * @param sh the current shell
34       */
35
36      public void doIt( StringTokenizer args, Shell sh )
37      {
38          // Get command keys from global table, print them out,
39          // followed by command's help string.
40
41          sh.getConsole().println( "shell commands" );
42          ShellCommandTable table = sh.getSystem().getCommandTable();
43          String[] names = table.getCommandNames();
44          for (int i = 0; i < names.length; i++ ) {
45              String cmdname = names[i];
46              ShellCommand cmd = table.lookup( cmdname );
47              sh.getConsole().
48                  println( "  " + cmdname + ": " + cmd.getHelpString() );
49          }
50      }
51  }
```

```
 1  // joi/6/juno/NewfileCommand.java
 2  //
 3  //
 4  // Copyright 2003, Ethan Bolker and Bill Campbell
 5
 6  import java.util.*;
 7
 8  /**
 9   * The Juno shell command to create a text file.
10   * Usage:
11   * <pre>
12   *     newfile filename contents
13   * </pre>
14   *
15   * @version 6
16   */
17
18  public class NewfileCommand extends ShellCommand
19  {
20      /**
21       * Construct a NewfileCommand object.
22       */
23
24      public NewfileCommand()
25      {
26          super( "create a new TextFile", "filename contents" );
27      }
28
29      /**
30       * Create a new TextFile in the current Directory.
31       *
32       * @param args the remainder of the command line.
33       * @param sh the current shell
34       */
35
36      public void doIt( StringTokenizer args, Shell sh )
37      {
38          String filename = args.nextToken();
39          String contents = args.nextToken("").trim(); // rest of line
40          new TextFile( filename, sh.getUser(), sh.getDot(), contents );
41      }
42  }
```

```
// joi/6/juno/ShellCommandTable.java (version 6)
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * A ShellCommandTable object maintains a dispatch table of
 * ShellCommand objects keyed by the command names used to invoke
 * them.
 *
 * To add a new shell command to the table, install it from
 * method fillTable().
 *
 * @see ShellCommand
 *
 * @version 6
 */

public class ShellCommandTable
{
    private Map table = new TreeMap();

    /**
     * Construct and fill a shell command table.
     */

    public ShellCommandTable()
    {
        fillTable();
    }

    /**
     * Get a ShellCommand, given the command name key.
     *
     * @param key the name associated with the command we're
     *            looking for.
     *
     * @return the command we're looking for, null if none.
     */

    public ShellCommand lookup( String key )
    {
        return (ShellCommand)table.get( key );
    }

    /**
     * Get an array of the command names.
     *
     * @return the array of command names.
     */

    public String[] getCommandNames()
    {
        return (String[]) table.keySet().toArray( new String[0] );
```



```
    }

    // Associate a command name with a ShellCommand.

    private void install( String commandName, ShellCommand command )
    {
        table.put( commandName, command );
    }

    // Fill the dispatch table with ShellCommands, keyed by their
    // command names.

    private void fillTable()
    {
        install( "newfile", new NewfileCommand() );
        install( "type", new TypeCommand() );
        install( "mkdir", new MkdirCommand() );
        install( "help", new HelpCommand() );
    }
}
```

```
// joi/6/jfiles/JFile.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.Date;
import java.io.File;

/**
 * A JFile object models a file in a hierarchical file system.
 * <p>
 * Extend this abstract class to create particular kinds of JFiles,
 * e.g.:<br>
 * Directory  -
 *    a JFile that maintains a list of the files it contains.<br>
 * TextFile  -
 *    a JFile containing text you might want to read.<br>
 *
 * @see Directory
 * @see TextFile
 *
 * @version 6
 */

public abstract class JFile
{
    /**
     * The separator used in pathnames.
     */

    public static final String separator = File.separator;

    private String    name;       // a JFile knows its name
    private User      owner;      // the owner of this file
    private Date      createDate; // when this file was created
    private Date      modDate;    // when this file was last modified
    private Directory parent;     // the Directory containing this file

    /**
     * Construct a new JFile, set owner, parent, creation and
     * modification dates. Add this to parent (unless this is the
     * root Directory).
     *
     * @param name     the name for this file (in its parent directory).
     * @param creator  the owner of this new file.
     * @param parent   the Directory in which this file lives.
     */

    protected JFile( String name, User creator, Directory parent )
    {
        this.name   = name;
        this.owner  = creator;
        this.parent = parent;
        if (parent != null) {
            parent.addJFile( name, this );
        }
```



```
        createDate  = modDate = new Date(); // set dates to now
    }

    /**
     * The name of the file.
     *
     * @return the file's name.
     */

    public String getName()
    {
        return name;
    }

    /**
     * The full path to this file.
     *
     * @return the path name.
     */

    public String getPathName()
    {
        if (this.isRoot()) {
            return separator;
        }
        if (parent.isRoot()) {
            return separator + getName();
        }
        return parent.getPathName() + separator + getName();
    }

    /**
     * The size of the JFile
     * (as defined by the child class)..
     *
     * @return the size.
     */

    public abstract int getSize();

    /**
     * Suffix used for printing file names
     * (as defined by the child class).
     *
     * @return the file's suffix.
     */

    public abstract String getSuffix();

    /**
     * Set the owner for this file.
     *
     * @param owner the new owner.
     */

    public void setOwner( User owner )
```

```
113      {
114          this.owner = owner;
115      }
116
117      /**
118       * The file's owner.
119       *
120       * @return the owner of the file.
121       */
122
123      public User getOwner()
124      {
125          return owner;
126      }
127
128      /**
129       * The date and time of the file's creation.
130       *
131       * @return the file's creation date and time.
132       */
133
134      public String getCreateDate()
135      {
136          return createDate.toString();
137      }
138
139      /**
140       * Set the modification date to "now".
141       */
142
143      protected void setModDate()
144      {
145          modDate = new Date();
146      }
147
148      /**
149       * The date and time of the file's last modification.
150       *
151       * @return the date and time of the file's last modification.
152       */
153
154      public String getModDate()
155      {
156          return modDate.toString();
157      }
158
159      /**
160       * The Directory containing this file.
161       *
162       * @return the parent directory.
163       */
164
165      public Directory getParent()
166      {
167          return parent;
168      }
```



```
169
170      /**
171       * A JFile whose parent is null is defined to be the root
172       * (of a tree).
173       *
174       * @return true when this JFile is the root.
175       */
176
177      public boolean isRoot()
178      {
179          return (parent == null);
180      }
181
182      /**
183       * How a JFile represents itself as a String.
184       * That is,
185       * <pre>
186       *    owner   size    modDate    name+suffix
187       * </pre>
188       *
189       * @return the String representation.
190       */
191
192      public String toString()
193      {
194          return getOwner() + "\t" +
195              getSize()  + "\t" +
196              getModDate()  + "\t" +
197              getName() + getSuffix();
198      }
199  }
```

```
// joi/6/jfiles/Directory.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Directory of JFiles.
 *
 * A Directory is a JFile that maintains a
 * table of the JFiles it contains
 *
 * @version 6
 */

public class Directory extends JFile
{
    private TreeMap jfiles;  // table for JFiles in this Directory

    /**
     * Construct a Directory.
     *
     * @param name     the name for this Directory (in its parent Directo
     * @param creator  the owner of this new Directory
     * @param parent   the Directory in which this Directory lives.
     */

    public Directory( String name, User creator, Directory parent)
    {
        super( name, creator, parent );
        jfiles = new TreeMap();
    }

    /**
     * The size of a directory is the number of TextFiles it contains.
     *
     * @return the number of TextFiles.
     */

    public int getSize()
    {
        return jfiles.size();
    }

    /**
     * Suffix used for printing Directory names;
     * we define it as the (system dependent)
     * name separator used in path names.
     *
     * @return the suffix for Directory names.
     */

    public String getSuffix()
    {
        return  JFile.separator;
```



```
    }

    /**
     * Add a JFile to this Directory. Overwrite if a JFile
     * of that name already exists.
     *
     * @param name the name under which this JFile is added.
     * @param afile the JFile to add.
     */

    public void addJFile(String name, JFile afile)
    {
        jfiles.put( name, afile );
        setModDate();
    }

    /**
     * Get a JFile in this Directory, by name .
     *
     * @param filename the name of the JFile to find.
     * @return the JFile found.
     */

    public JFile retrieveJFile( String filename )
    {
        JFile aFile = (JFile)jfiles.get( filename );
        return aFile;
    }

    /**
     * Get the contents of this Directory as an array of
     * the file names, each of which is a String.
     *
     * @return the array of names.
     */

    public String[] getFileNames()
    {
        return (String[])jfiles.keySet().toArray( new String[0] );
    }
}
```

```
// joi/6/jfiles/TextFile.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * A TextFile is a JFile that holds text.
 *
 * @version 6
 */

public class TextFile extends JFile
{
    private String contents;  // The text itself

    /**
     * Construct a TextFile with initial contents.
     *
     * @param name    the name for this TextFile (in its parent Directory
     * @param creator the owner of this new TextFile
     * @param parent  the Directory in which this TextFile lives.
     * @param initialContents the initial text
     */

    public TextFile( String name, User creator, Directory parent,
              String initialContents )
    {
        super( name, creator, parent );
        setContents( initialContents );
    }

    /**
     * Construct an empty TextFile.
     *
     * @param name    the name for this TextFile (in its parent Directory
     * @param creator the owner of this new TextFile
     * @param parent  the Directory in which this TextFile lives
     */

    TextFile( String name, User creator, Directory parent )
    {
        this( name, creator, parent, "" );
    }

    /**
     * The size of a text file is the number of characters stored.
     *
     * @return the file's size.
     */

    public int getSize()
    {
        return contents.length();
    }

    /**
```



```
     * Suffix used for printing text file names is "".
     *
     * @return an empty suffix (for TextFiles).
     */

    public String getSuffix()
    {
        return "";
    }

    /**
     * Replace the contents of the file.
     *
     * @param contents the new contents.
     */

    public void setContents( String contents )
    {
        this.contents = contents;
        setModDate();
    }

    /**
     * The contents of a text file.
     *
     * @return String contents of the file.
     */

    public String getContents()
    {
        return contents;
    }

    /**
     * Append text to the end of the file.
     *
     * @param  text the text to be appended.
     */

    public void append( String text )
    {
        setContents( contents + text );
    }


    /**
     * Append a new line of text to the end of the file.
     *
     * @param  text the text to be appended.
     */

    public void appendLine( String text )
    {
        this.setContents(contents + '\n' + text);
    }
}
```

```
 1  // joi/6/juno/User.java
 2  //
 3  //
 4  // Copyright 2003 Ethan Bolker and Bill Campbell
 5
 6  /**
 7   * Model a Juno user.  Each User has a login name,
 8   * a home directory, and a real name.
 9   *
10   * @version 6
11   */
12
13  public class User
14  {
15      private String name;     // the User's login name
16      private Directory home;  // her home Directory
17      private String realName; // her real name
18
19      /**
20       * Construct a new User.
21       *
22       * @param name     the User's login name.
23       * @param home     her home Directory.
24       * @param realName her real name.
25       */
26
27      public User( String name, Directory home, String realName )
28      {
29          this.name     = name;
30          this.home     = home;
31          this.realName = realName;
32      }
33
34      /**
35       * Get the User's login name.
36       *
37       * @return the name.
38       */
39
40      public String getName()
41      {
42          return name;
43      }
44
45      /**
46       * Convert the User to a String.
47       * The String representation for a User is her
48       * login name.
49       *
50       * @return the User's name.
51       */
52
53      public String toString()
54      {
55          return getName();
56      }
```



```
57
58      /**
59       * Get the User's home Directory.
60       *
61       * @return the home Directory.
62       */
63
64      public Directory getHome()
65      {
66          return home;
67      }
68
69      /**
70       * Get the user's real name.
71       *
72       * @return the real name.
73       */
74
75      public String getRealName()
76      {
77          return realName;
78      }
79  }
```