

```

1 // foj/10/Juno/Juno.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.io.*;
7 import java.util.*;
8 import java.lang.*;
9
10 /**
11  * Juno (Juno's Unix NOC) mimics a command line operating system
12  * such as Unix.
13  * <p>
14  * A Juno system has a name, a set of users, a JFile system,
15  * a login process and a set of shell commands.
16  *
17  * @see User
18  * @see JFile
19  * @see ShellCommand
20  *
21  * @version 10
22  */
23
24 public class Juno
25     implements Serializable
26 {
27     private final static String OS      = "Juno";
28     private final static String VERSION = "10";
29
30     private String  hostname; // host machine name
31     private Map    users; // lookup table for users
32     private transient OutputInterface console;
33
34     private Directory slash; // root of JFile system
35     private Directory userHomes; // for home directories
36
37     private ShellCommandTable commandTable; // shell commands
38
39     // file containing Juno state
40     private transient String fileName = null;
41
42     // port used by Juno server for remote login
43     private int junoPort = 2001;
44
45     /**
46      * Construct a Juno (operating system) object.
47      *
48      * @param hostname the name of the host on which it's running.
49      * @param echoInput should all input be echoed as output?
50      * @param isGUI graphical user interface?
51      * @param isRemote running as a server?
52      */
53
54     public Juno( String hostname, boolean echoInput,
55                 boolean isGUI, boolean isRemote )
56 {

```

```

57 // Initialize the Juno environment ...
58 this.hostname      = hostname;
59 users              = new TreeMap();
60 commandTable      = new ShellCommandTable();
61
62 // the file system
63
64 slash = new Directory( "", null, null );
65 User root = new User( "root", "swordfish", slash,
66                      "Rick Martin" );
67 users.put( "root", root );
68 slash.setOwner( root );
69 userHomes = new Directory( "users", root, slash );
70
71 }
72
73 // Set up the correct console:
74 // command line (default), graphical or remote.
75
76 private void setupConsole( boolean echoInput, boolean isGUI,
77                            boolean isRemote )
78 {
79     LoginInterpreter interpreter
80         = new LoginInterpreter( this, null );
81
82     if ( isGUI ) {
83         console = new GUILoginConsole( hostname,
84                                       this, interpreter, echoInput );
85     }
86     else if ( isRemote ) {
87         console = new RemoteConsole( this, echoInput, junoPort );
88     }
89     else {
90         console = new JunoTerminal( echoInput );
91     }
92
93     // Tell the interpreter about the console
94     interpreter.setConsole( console );
95
96     // If we're using a simple command line interface,
97     // start that. (Constructing a GUI starts the GUI.)
98     // Shut down Juno when done
99
100    if ( !isGUI && !isRemote ) {
101        interpreter.CLIlogin();
102        shutdown();
103    }
104
105    /**
106     * Shut down this Juno system.
107     *
108     * Save state if required.
109     */
110
111    public void shutdown( )
112 {

```

```

113     {
114         if (fileName != null) {
115             writeJuno( );
116         }
117     }
118 }
119 /**
120  * Set the name of file in which system state is kept.
121  *
122  * @param fileName the file name.
123  */
124
125 public void setFileName(String fileName)
126 {
127     this.fileName = fileName;
128 }
129
130 /**
131  * The name of the host computer on which this system
132  * is running.
133  *
134  * @return the host computer name.
135  */
136
137 public String getHostName()
138 {
139     return hostName;
140 }
141
142 /**
143  * The name of this operating system.
144  *
145  * @return the operating system name.
146  */
147
148 public String getOS()
149 {
150     return OS;
151 }
152
153 /**
154  * The version number for this system.
155  *
156  * @return the version number.
157  */
158
159 public String getVersion()
160 {
161     return VERSION;
162 }
163
164 /**
165  * The directory containing all user homes for this system.
166  *
167  * @return the directory containing user homes.
168  */

```

```

169
170 public Directory getUserHomes()
171 {
172     return userHomes;
173 }
174
175 /**
176  * The shell command table for this system.
177  *
178  * @return the shell command table.
179  */
180
181 public ShellCommandTable getCommandTable()
182 {
183     return commandTable;
184 }
185
186 /**
187  * Look up a user by user name.
188  *
189  * @param username the user's name.
190  * @return the appropriate User object.
191  */
192
193 public User lookupUser( String username )
194 {
195     return (User) users.get( username );
196 }
197
198 /**
199  * Create a new User.
200  *
201  * @param userName the User's login name.
202  * @param home her home Directory.
203  * @param password her password.
204  * @param realName her real name.
205  * @return newly created User.
206  */
207
208 public User createUser( String userName, Directory home,
209                        String password, String realName )
210 {
211     User newUser = new User( userName, password,
212                             home, realName );
213     users.put( userName, newUser );
214     return newUser;
215 }
216
217 /**
218  * The Juno system may be given the following command line
219  * arguments:
220  *
221  * -e:      Echo all input (useful for testing).
222  *
223  * -version: Report the version number and exit.
224  */

```

```

225 * -g:          Support a GUI console.
226 *           Start Juno server.
227 * -remote
228 *           -f filename file to read/write system state from/to
229 *
230 * [hostname]: The name of the host on which
231 *             Juno is running (optional).
232 * /
233
234
235 public static void main( String[] args )
236 {
237     // Parse command line options
238
239     boolean echoInput    = false;
240     boolean versionQuery = false;
241     boolean isGUI        = false;
242     boolean isRemote     = false;
243     String  hostname     = "mars";
244     String  junoFileName = null;
245
246     for (int i=0; i < args.length; i++) {
247         if (args[i].equals("-e")) {
248             echoInput = true;
249         }
250         else if (args[i].equals("-version")) {
251             versionQuery = true;
252         }
253         else if (args[i].equals("-g")) {
254             isGUI = true;
255         }
256         else if (args[i].equals("-remote" )) {
257             isRemote = true;
258         }
259         else if (args[i].equals("-f")) {
260             junoFileName = args[i+1];
261         }
262         else {
263             hostname = args[i];
264         }
265     }
266
267     // If it's a version query give the version and exit
268     if ( versionQuery ) {
269         System.out.println( OS + " version " + VERSION );
270         System.exit(0);
271     }
272
273     // Create a new Juno or read one from a file.
274     Juno junoSystem = null;
275     if (junoFileName != null) {
276         junoSystem = readJuno( junoFileName );
277     }
278     if (junoSystem == null) {
279         junoSystem = new Juno(  hostname, echoInput,
280                               isGUI, isRemote );

```

```

281     }
282     junoSystem.setFileName( junoFileName );
283     junoSystem.setupConsole( echoInput, isGUI, isRemote );
284 }
285
286 // Read Juno state from a file.
287 //
288 // @param junoFileName the name of the file containing the system.
289 // @return the system, null if file does not exist.
290
291 private static Juno readJuno( String junoFileName )
292 {
293     File file = new File( junoFileName );
294     if (!file.exists()) {
295         return null;
296     }
297     ObjectInputStream instream = null;
298     try {
299         instream = new ObjectInputStream(
300             new FileInputStream( file ) );
301         Juno juno = (Juno) instream.readObject();
302         System.out.println(
303             "Juno state read from file " + junoFileName);
304         return juno;
305     }
306     catch (Exception e ) {
307         System.err.println("Problem reading " + junoFileName );
308         System.err.println(e);
309         System.exit(1);
310     }
311     finally {
312         try {
313             instream.close();
314         }
315         catch (Exception e) {
316         }
317     }
318     return null; // you can never get here
319 }
320
321 // Write Juno state to a file.
322 private void writeJuno()
323 {
324     ObjectOutputStream outputStream = null;
325     try {
326         outputStream = new ObjectOutputStream(
327             new FileOutputStream( fileName ) );
328         outputStream.writeObject( this );
329         System.out.println(
330             "Juno state written to file " + fileName);
331     }
332     catch (Exception e ) {
333         System.err.println("Problem writing " + fileName);
334         System.err.println(e);
335     }
336 }

```

```
337         finally {
338             try {
339                 outputStream.close();
340             }
341             catch (Exception e ) {
342             }
343         }
344     }
345 }
```

```

1 // foj/10/Juno/LoginInterpreter.java
2 //
3 //
4 // Copyright 2003 Ethan Bolker and Bill Campbell
5
6 import java.util.*;
7
8 /**
9  * Interpreter for Juno login commands.
10 *
11 * There are so few commands that if-then-else logic is OK.
12 *
13 * @version 1.0
14 */
15
16 public class LoginInterpreter
17     implements InterpreterInterface
18 {
19     private static final String LOGIN_COMMANDS =
20         "help, register, <username>, exit";
21
22     private Juno system; // the Juno object
23     private OutputInterface console; // where output goes
24
25     /**
26      * Construct a new LoginInterpreter for interpreting
27      * login commands.
28      *
29      * @param system the system creating this interpreter.
30      * @param console the terminal used for input and output.
31      */
32
33     public LoginInterpreter( Juno system, OutputInterface console)
34     {
35         this.system = system;
36         this.console = console;
37     }
38
39     /**
40      * Set the console for this interpreter. Used by the
41      * creator of this interpreter.
42      *
43      * @param console the Terminal to be used for input and output.
44      */
45
46     public void setConsole( OutputInterface console)
47     {
48         this.console = console;
49     }
50
51     /**
52      * Simulates behavior at login: prompt.
53      */
54
55     public void CLILogin()
56     {

```

```

57         welcome();
58         boolean moreWork = true;
59         while( moreWork ) {
60             moreWork = interpret(((InputInterface)console).
61                 readLine( "Juno login: " ) );
62         }
63     }
64
65     /**
66      * Parse user's command line and dispatch appropriate
67      * semantic action.
68      *
69      * @param inputLine the User's instructions.
70      * @return true except for "exit" command
71      *         or null inputLine.
72      */
73
74     public boolean interpret( String inputLine )
75     {
76         if (inputLine == null) {
77             return false;
78         }
79         StringTokenizer st =
80             new StringTokenizer( inputLine );
81         if (st.countTokens() == 0) {
82             return true; // skip blank line
83         }
84         String visitor = st.nextToken();
85         if (visitor.equals( "exit" )) {
86             return false;
87         }
88         if (visitor.equals( "register" )) {
89             register( st );
90         }
91         else if (visitor.equals( "help" )) {
92             help();
93         }
94         else {
95             String password;
96             try {
97                 if (console.isGUI()) {
98                     password = st.nextToken();
99                 }
100                else {
101                    password = readPassword( "password: " );
102                }
103            }
104            User user = system.lookupUser( visitor );
105            user.matchPassword( password );
106            new Shell( system, user, console );
107        }
108        catch (Exception e) {
109
110            // NullPointerException if no such user,
111            // IOException if password fails to match -
112            // message to user doesn't give away which.

```

```

113 // The sysadmin would probably want a log
114 // that did keep track.
115 //
116 // Other exceptions should be caught in shell()
117 console.println("sorry");
118 }
119 }
120 return true;
121 }
122 }
123 // Register a new user, giving him or her a login name and a
124 // home directory on the system.
125 //
126 // StringTokenizer argument contains the new user's login name
127 // followed by full real name.
128 private void register( StringTokenizer line )
129 {
130     String username = "";
131     String password = "";
132     String realname = "";
133     try {
134         username = line.nextToken();
135         password = line.nextToken();
136         realname = line.nextToken().trim();
137     } catch (NoSuchElementException e) {
138     }
139     if (username.equals("") || password.equals("") ||
140         realname.equals("")) {
141         console.println(
142             "please supply username, password, real name");
143         return;
144     }
145     User user = system.lookupUser(username);
146     if (user != null) { // user already exists
147         console.println("sorry");
148         return;
149     }
150     if (badPassword( password )) {
151         console.println("password too easy to guess");
152         return;
153     }
154     Directory home = new Directory( username, null,
155         system.getUserHomes() );
156     user = system.createUser( username, home, password, realname );
157     home.setOwner( user );
158 }
159 // test to see if password is unacceptable:
160 // fewer than 6 characters
161 // contains only alphabetic characters
162
163
164
165
166
167
168

```

```

169 private boolean badPassword( String pwd )
170 {
171     if (pwd.length() < 6) {
172         return true;
173     }
174     int nonAlphaCount = 0;
175     for (int i=0; i < pwd.length(); i++) {
176         if (!Character.isLetter(pwd.charAt(i))) {
177             nonAlphaCount++;
178         }
179     }
180     return (nonAlphaCount == 0);
181 }
182 // Used for reading the user's password in CLI.
183 private String readPassword( String prompt )
184 {
185     String line =
186         ((InputInterface) console).readline( prompt );
187     StringTokenizer st = new StringTokenizer( line );
188     try {
189         return st.nextToken();
190     } catch ( NoSuchElementException e ) {
191         return ""; // keeps compiler happy
192     }
193 }
194 // Display a short welcoming message, and remind users of
195 // available commands.
196 private void welcome()
197 {
198     console.println( "Welcome to " + system.getHostname() +
199         " running " + system.getOS() +
200         " version " + system.getVersion() );
201     help();
202 }
203 // Remind user of available commands.
204 private void help()
205 {
206     console.println( LOGIN_COMMANDS );
207     console.println("");
208 }
209 }
210
211
212
213
214
215
216
217

```

```

1 // foj/10/Juno/Shell.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
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 1.0
18 */
19
20 public class Shell
21 implements InterpreterInterface
22 {
23     private Juno system; // The operating system object
24     private User user; // The user logged in
25     private OutputInterface console; // The console for this shell
26     private Directory dot; // The current working directory
27
28     /**
29      * Construct a login shell for the given user and console.
30      *
31      * @param system a reference to the Juno system.
32      * @param user the User logging in.
33      * @param console a Terminal for input and output.
34      */
35
36     Shell( Juno system, User user, OutputInterface console )
37     {
38         this.system = system;
39         this.user = user;
40         this.console = console;
41         dot = user.getHome(); // default current directory
42
43         if (!console.isGUI()) {
44             this.console = console;
45             CLIShell();
46         }
47         else
48             this.console =
49                 new GUIShellConsole("Juno shell for " + user,
50                                     this, console.isEchoInput());
51     }
52
53     // Run the command line interpreter
54     private void CLIShell()
55     {
56

```

```

57     boolean moreWork = true;
58     while(moreWork) {
59         moreWork = interpret( ((InputInterface) console).
60                               readline( getPrompt() ) );
61     }
62     console.println("goodbye");
63 }
64
65 /**
66  * Interpret a String.
67  *
68  * Syntax
69  * <pre>
70  * shellcommand command-arguments
71  * </pre>
72  *
73  * @param inputLine the String to interpret.
74  * @return true unless shell command is logout.
75  */
76
77     public boolean interpret( String inputLine )
78     {
79         StringTokenizer st = stripComments(inputLine);
80         if (st.countTokens() == 0) { // skip blank line
81             return true;
82         }
83         String commandName = st.nextToken();
84         ShellCommand commandObject =
85             system.getCommandTable().lookup( commandName );
86         if (commandObject == null ) {
87             console.errPrintln( "Unknown command: " + commandName );
88             return true;
89         }
90         try {
91             commandObject.doIt( st, this );
92         }
93         catch (ExitShellException e) {
94             return false;
95         }
96         catch (BadShellCommandException e) {
97             console.errPrintln( "Usage: " + commandName + " " +
98                                 e.getCommand().getArgString() );
99         }
100         catch (JunoException e) {
101             console.errPrintln( e.getMessage() );
102         }
103         catch (Exception e) {
104             console.errPrintln( "you should never get here" );
105             console.errPrintln( e.toString() );
106         }
107         return true;
108     }
109
110     // Strip characters from '#' to end of line, create and
111     // return a StringTokenizer for what's left.
112

```

```

113 private StringTokenizer stripComments( String line )
114 {
115     int commentIndex = line.indexOf('#');
116     if (commentIndex >= 0) {
117         line = line.substring(0,commentIndex);
118     }
119     return new StringTokenizer(line);
120 }
121
122 /**
123  * The prompt for the CLI.
124  */
125 * @return the prompt string.
126 */
127
128 public String getPrompt()
129 {
130     return system.getHostname() + ":" + getDot().getPathName() + "> ";
131 }
132
133 /**
134  * The User associated with this shell.
135  */
136 * @return the user.
137 */
138
139 public User getUser()
140 {
141     return user;
142 }
143
144 /**
145  * The current working directory for this shell.
146  */
147 * @return the current working directory.
148 */
149
150 public Directory getDot()
151 {
152     return dot;
153 }
154
155 /**
156  * Set the current working directory for this Shell.
157  */
158 * @param dot the new working directory.
159 */
160
161 public void setDot(Directory dot)
162 {
163     this.dot = dot;
164 }
165
166 /**
167  * The console associated with this Shell.
168

```

```

169     *
170     * @return the console.
171     */
172     public OutputInterface getConsole()
173     {
174         return console;
175     }
176 }
177
178 /**
179  * The Juno object associated with this Shell.
180  */
181 * @return the Juno instance that created this Shell.
182 */
183
184 public Juno getSystem()
185 {
186     return system;
187 }
188 }

```



```

1 // fo1/10/juno/ShellCommand.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
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 10
15 */
16
17 public abstract class ShellCommand
18     implements java.io.Serializable
19 {
20     private String helpString; // documents the command
21     private String argString; // any args to the command
22
23     /**
24      * A constructor, always called (as super()) by the subclass.
25      * Used only for commands that have arguments.
26      *
27      * @param helpString a brief description of what the command does.
28      * @param argString a prototype illustrating the required arguments.
29      */
30
31     protected ShellCommand( String helpString, String argString )
32     {
33         this.argString = argString;
34         this.helpString = helpString;
35     }
36
37     /**
38      * A constructor for commands having no arguments.
39      *
40      * @param helpString a brief description of what the command does.
41      */
42
43     protected ShellCommand( String helpString )
44     {
45         this( helpString, "" );
46     }
47
48     /**
49      * Execute the command.
50      *
51      * @param args the remainder of the command line.
52      * @param sh the current shell
53      *
54      * @exception JunoException for reporting errors
55      */
56

```

```

57     public abstract void doIt( StringTokenizer args, Shell sh )
58         throws JunoException;
59
60     /**
61      * Help for this command.
62      *
63      * @return the help string.
64      */
65
66     public String getHelpString()
67     {
68         return helpString;
69     }
70
71     /**
72      * The argument string prototype.
73      *
74      * @return the argument string prototype.
75      */
76
77     public String getArgString()
78     {
79         return argString;
80     }
81 }

```

```

1 // foj/10/juno/ShellCommandTable.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * A ShellCommandTable object maintains a dispatch table of
10 * ShellCommand objects keyed by the command names used to invoke
11 * them.
12 *
13 * To add a new shell command to the table, install it from
14 * method fillTable().
15 *
16 * @see ShellCommand
17 *
18 * @version 10
19 */
20
21 public class ShellCommandTable
22     implements java.io.Serializable
23 {
24     private Map table = new TreeMap();
25
26     /**
27      * Construct and fill a shell command table.
28      */
29
30     public ShellCommandTable()
31     {
32         fillTable();
33     }
34
35     /**
36      * Get a ShellCommand, given the command name key.
37      *
38      * @param key the name associated with the command we're
39      *         looking for.
40      *
41      * @return the command we're looking for, null if none.
42      */
43
44     public ShellCommand lookup( String key )
45     {
46         ShellCommand commandObject = (ShellCommand) table.get( key );
47         if (commandObject != null) {
48             return commandObject;
49         }
50
51         // try to load dynamically
52         // construct classname = "KeyCommand"
53         char[] chars = (key + "Command").toCharArray();
54         chars[0] = key.toUpperCase().charAt(0);
55         String classname = new String(chars);
56         try {

```

```

57         commandObject =
58             (ShellCommand)Class.forName(classname).newInstance();
59     }
60     catch (Exception e) { // couldn't find class
61         return null;
62     }
63     install(key, commandObject); // put it in table for next time
64     return commandObject;
65 }
66
67 /**
68  * Get an array of the command names.
69  *
70  * @return the array of command names.
71  */
72
73     public String[] getCommandNames()
74     {
75         return (String[]) table.keySet().toArray( new String[0] );
76     }
77
78     // Associate a command name with a ShellCommand.
79
80     private void install( String commandName, ShellCommand command )
81     {
82         table.put( commandName, command );
83     }
84
85     // Fill the dispatch table with ShellCommands, keyed by their
86     // command names.
87
88     private void fillTable()
89     {
90         install( "list", new ListCommand() );
91         install( "cd", new CdCommand() );
92         install( "newfile", new NewFileCommand() );
93         install( "remove", new RemoveCommand() );
94         install( "help", new HelpCommand() );
95         install( "mkdir", new MkdirCommand() );
96         install( "type", new TypeCommand() );
97         install( "logout", new LogoutCommand() );
98     }
99 }

```

```
1 // fo1/10/juno/MkdirCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
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 1.0
16  */
17
18 public class MkdirCommand extends ShellCommand
19 {
20     MkdirCommand()
21     {
22         super( "create a subdirectory of the current directory",
23             "directory-name" );
24     }
25
26     /**
27      * Create a new Directory in the current Directory.
28      *
29      * @param args the remainder of the command line.
30      * @param sh the current shell.
31      *
32      * @exception JunoException for reporting errors.
33      */
34
35     public void doit( StringTokenizer args, Shell sh )
36     {
37         throws JunoException
38     {
39         String filename = args.nextToken();
40         new Directory( filename, sh.getUser(), sh.getDot() );
41     }
42 }
```

```

1 // fo1/10/juno/TypeCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
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 1.0
17 */
18
19 public class TypeCommand extends ShellCommand
20 {
21     TypeCommand()
22     {
23         super( "display contents of a TextFile", "textfile" );
24     }
25
26     /**
27     * Display the contents of a TextFile.
28     *
29     * @param args the remainder of the command line.
30     * @param sh the current Shell
31     *
32     * @exception JunoException for reporting errors
33     */
34
35     public void doIt( StringTokenizer args, Shell sh )
36     throws JunoException
37     {
38         String filename;
39
40         try {
41             filename = args.nextToken();
42         }
43         catch (NoSuchElementException e) {
44             throw new BadShellCommandException( this );
45         }
46         try {
47             sh.getConsole().println(
48                 ( (TextFile) sh.getDot() ).
49                 retrieveFile( filename ) ).getContents();
50         }
51         catch (NullPointerException e) {
52             throw new JunoException( "JFile does not exist: "
53                 + filename);
54         }
55         catch (ClassCastException e) {
56             throw new JunoException( "JFile not a text file: "

```

```

57     }
58     }
59 }

```

```
1 // fo1/10/juno/HelpCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
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 1.0
16  */
17
18 public class HelpCommand extends ShellCommand
19 {
20     HelpCommand()
21     {
22         super( "display ShellCommands" );
23     }
24
25     /**
26      * Print out help for all commands.
27      *
28      * @param args the remainder of the command line.
29      * @param sh the current shell
30      *
31      * @exception JunoException for reporting errors
32      */
33
34     public void doIt( StringTokenizer args, Shell sh )
35     {
36         throws JunoException
37     {
38         // Get command keys from global table, print them out.
39
40         sh.getConsole().println( "shell commands" );
41         ShellCommandTable table = sh.getSystem().getCommandTable();
42         String[] names = table.getCommandNames();
43         for (int i = 0; i < names.length; i++) {
44             String cmdname = names[i];
45             ShellCommand cmd =
46                 (ShellCommand) table.lookup( cmdname );
47             sh.getConsole().
48                 println( " " + cmdname + " : " + cmd.getHelpString() );
49         }
50     }
51 }
```



```
1 // fo1/10/juno/CdCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * The Juno shell command to change directory.
10  * Usage:
11  * <pre>
12  *   cd [directory]
13  * </pre>
14  * For moving to the named directory.
15  *
16  * @version 1.0
17  */
18
19 class CdCommand extends ShellCommand
20 {
21     CdCommand()
22     {
23         super( "change current directory", "[ directory ]" );
24     }
25
26     /**
27      * Move to the named directory
28      *
29      * @param args the remainder of the command line.
30      * @param sh the current shell
31      *
32      * @exception JunoException for reporting errors
33      */
34
35     public void doIt( StringTokenizer args, Shell sh )
36         throws JunoException
37     {
38         String dirname = "";
39         Directory d = sh.getUser().getHome(); // default
40         if ( args.hasMoreTokens() ) {
41             dirname = args.nextToken();
42             if (dirname.equals(".")) {
43                 if (sh.getDot().isRoot())
44                     d = sh.getDot(); // no change
45                 else
46                     d = sh.getDot().getParent();
47             }
48             else if (dirname.equals("..")) {
49                 d = sh.getDot(); // no change
50             }
51             else {
52                 d = (Directory)(sh.getDot().retrieveFile(dirname));
53             }
54         }
55         sh.setDot( d );
56     }
57 }
```

57 }

```
1 // fo1/10/Juno/ListCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * The Juno shell command to list contents of the current directory.
10  * Usage:
11  * <pre>
12  *     list
13  * </pre>
14  * @version 1.0
15  */
16
17 public class ListCommand extends ShellCommand
18 {
19     // The constructor adds this object to the global table.
20
21     ListCommand()
22     {
23         super( "list contents of current directory" );
24     }
25
26     /**
27      * List contents of the current working directory.
28      *
29      * @param args the remainder of the command line.
30      * @param sh   the current shell
31      *
32      * @exception JunoException for reporting errors
33      */
34
35     public void dotL( StringTokenizer args, Shell sh )
36     throws JunoException
37     {
38         OutputInterface terminal = sh.getConsole();
39         Directory dir           = sh.getDot();
40         String[] fileNames     = dir.getFileNames();
41
42         terminal.println( dir.getPathName() );
43         for ( int i = 0; i < fileNames.length; i++ ) {
44             String fileName = fileNames[i];
45             JFile jfile     = dir.retrieveJFile( fileName );
46             terminal.println( jfile.toString() );
47         }
48     }
49 }
50
```



```

1 // foj/10/juno/GetfileCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7 import java.io.*;
8
9 /**
10  * The Juno shell command to get a text file from the underlying
11  * operating system and copy it to a Juno text file.
12  * Usage:
13  * <pre>
14  *   getfile native-filename juno-filename
15  * </pre>
16  * <pre>
17  *   *
18  *   * @version 10
19  *   * /
20  */
21
22 class GetfileCommand extends ShellCommand
23 {
24     GetfileCommand()
25     {
26         super( "download a file to Juno",
27              "native-filename juno-filename" );
28     }
29
30     /**
31      * Use the getfile command to copy the content of a real
32      * file to a Juno TextFile.
33      * <p>
34      * The command has the form:
35      * <pre>
36      *   get nativefile textfile <k>
37      *
38      * @param args: the remainder of the command line.
39      * @param sh: the current shell
40      *
41      * @exception JunoException for reporting errors
42      */
43
44     public void doIt( StringTokenizer args, Shell sh )
45     {
46         throws JunoException
47     {
48         if ( sh.getConsole().isRemote() ) {
49             throw( new JunoException(
50                 "Get not implemented for remote consoles." ) );
51         }
52         String src;
53         String dst;
54         try {
55             src = args.nextToken();
56             dst = args.nextToken();
57         }
58     }
59
60     }
61
62     }
63
64     }
65
66     }
67
68     }
69
70     }
71
72     }
73
74     }
75
76     }
77
78     }
79
80     }
81
82     }
83
84     }
85
86     }

```

```

57     catch (NoSuchElementException e) {
58         throw new BadShellCommandException( this );
59     }
60     BufferedReader instream = null;
61     Writer outstream = null;
62     try {
63         instream = new BufferedReader( new FileReader( src ) );
64         outstream = new StringWriter();
65         String line;
66         while ((line = instream.readLine()) != null) {
67             outstream.write( line );
68             outstream.write( '\n' );
69         }
70         new TextFile( dst, sh.getUser(),
71                     sh.getDot(), outstream.toString() );
72     }
73     catch (IOException e) {
74         throw new JunoException( "IO problem in get" );
75     }
76     finally {
77         try {
78             instream.close();
79             outstream.close();
80         }
81         catch (IOException e) {}
82     }
83 }
84 }
85 }

```

```
1 // fo1/10/juno/RemoveCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * The Juno shell command to remove a text file.
10  * Usage:
11  * <pre>
12  *     remove textfile
13  * </pre>
14  *
15  * @version 1.0
16  */
17
18 public class RemoveCommand extends ShellCommand
19 {
20     RemoveCommand()
21     {
22         super( "remove a TextFile", "textfile" );
23     }
24
25     /**
26      * Remove a TextFile.
27      *
28      * @param args the remainder of the command line.
29      * @param sh the current Shell
30      *
31      * @exception JunoException for reporting errors
32      */
33
34     public void doIt( StringTokenizer args, Shell sh )
35         throws JunoException
36     {
37         String filename = args.nextToken();
38         sh.getDot().removeFile(filename);
39     }
40 }
41
```

```
1 // foj/10/juno/LogoutCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * The Juno shell command to log out.
10  * Usage:
11  * <pre>
12  *   logout
13  * </pre>
14  *
15  * @version 10
16  */
17
18 public class LogoutCommand extends ShellCommand
19 {
20     LogoutCommand()
21     {
22         super( "log out, return to login: prompt" );
23     }
24
25     /**
26      * Log out from the current shell.
27      *
28      * @param args the remainder of the command line.
29      * @param sh the current shell
30      *
31      * @exception JunoException for reporting errors
32      */
33
34     public void doIt( StringTokenizer args, Shell sh )
35     {
36         throws JunoException
37         {
38             throw new ExitShellException();
39         }
40     }
41 }
```

```

1 // fo1/10/jfiles/JFile.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.util.Date;
7 import java.io.File;
8
9 /**
10 * A JFile object models a file in a hierarchical file system.
11 * <p>
12 * Extend this abstract class to create particular kinds of JFiles,
13 * e.g.:<br>
14 *   Directory _
15 *   * a JFile that maintains a list of the files it contains.<br>
16 *   * TextFile _
17 *   * a JFile containing text you might want to read.<br>
18 *
19 * @see Directory
20 * @see TextFile
21 *
22 * @version 10
23 */
24
25 public abstract class JFile
26     implements java.io.Serializable
27 {
28     /**
29      * The separator used in pathnames.
30      */
31
32     public static final String separator = File.separator;
33
34     private String name; // a JFile knows its name
35     private User owner; // the owner of this file
36     private Date createDate; // when this file was created
37     private Date modDate; // when this file was last modified
38     private Directory parent; // the Directory containing this file
39
40     /**
41      * Construct a new JFile, set owner, parent, creation and
42      * modification dates. Add this to parent (unless this is the
43      * root Directory).
44      *
45      * @param name the name for this file (in its parent directory).
46      * @param creator the owner of this new file.
47      * @param parent the Directory in which this file lives.
48      */
49
50     protected JFile( String name, User creator, Directory parent )
51     {
52         this.name = name;
53         this.owner = creator;
54         this.parent = parent;
55         if (parent != null) {
56             parent.addJFile( name, this );

```

```

57     }
58     createDate = modDate = new Date(); // set dates to now
59     }
60
61     /**
62      * The name of the file.
63      *
64      * @return the file's name.
65      */
66
67     public String getName()
68     {
69         return name;
70     }
71
72     /**
73      * The full path to this file.
74      *
75      * @return the path name.
76      */
77
78     public String getPathName()
79     {
80         if (this.isRoot()) {
81             return separator;
82         }
83         if (parent.isRoot()) {
84             return separator + getName();
85         }
86         return parent.getPathName() + separator + getName();
87     }
88
89     /**
90      * The size of the JFile
91      * (as defined by the child class)..
92      *
93      * @return the size.
94      */
95
96     public abstract int getSize();
97
98     /**
99      * Suffix used for printing file names
100      * (as defined by the child class).
101      *
102      * @return the file's suffix.
103      */
104
105     public abstract String getSuffix();
106
107     /**
108      * Set the owner for this file.
109      *
110      * @param owner the new owner.
111      */
112

```

```

113 public void setOwner( User owner )
114 {
115     this.owner = owner;
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 /**
172  * A JFile whose parent is null is defined to be the root
173  * (of a tree).
174  */
175 * @return true when this JFile is the root.
176 */
177
178 public boolean isRoot()
179 {
180     return (parent == null);
181 }
182
183 /**
184  * How a JFile represents itself as a String.
185  * That is,
186  * <pre>
187  *   owner      size      modDate      name+suffix
188  * </pre>
189  *
190  * @return the String representation.
191  */
192
193 public String toString()
194 {
195     return getOwner() + "\t" +
196         getSize() + "\t" +
197         getModDate() + "\t" +
198         getName() + getSuffix();
199 }
200 }

```

```

1 // fo1/10/juno/Directory.java
2 //
3 //
4 // Copyright 2003 Ethan Bolker and Bill Campbell
5
6 import java.util.*;
7
8 /**
9  * Directory of JFiles.
10
11  * A Directory is a JFile that maintains a
12  * table of the JFiles it contains.
13  *
14  * @version 10
15  */
16
17 public class Directory extends JFile
18 {
19     private TreeMap jfiles; // table for JFiles in this Directory
20
21     /**
22      * Construct a Directory.
23
24      * @param name the name for this Directory (in its parent Directory)
25      * @param creator the owner of this new Directory.
26      * @param parent the Directory in which this Directory lives.
27      */
28
29     public Directory( String name, User creator, Directory parent)
30     {
31         super( name, creator, parent );
32         jfiles = new TreeMap();
33     }
34
35     /**
36      * The size of a Directory is the number of JFiles it contains.
37
38      * @return the Directory's size.
39      */
40
41     public int getSize()
42     {
43         return jfiles.size();
44     }
45
46     /**
47      * Suffix used for printing Directory names;
48      * we define it as the (system dependent)
49      * name separator used in path names.
50
51      * @return the suffix for Directory names.
52      */
53
54     public String getSuffix()
55     {
56         return JFile.separator;

```

```

57     }
58
59     /**
60      * Add a JFile to this Directory. Overwrite if a JFile
61      * of that name already exists.
62
63      * @param name the name under which this JFile is added.
64      * @param afile the JFile to add.
65      */
66
67     public void addJFile( String name, JFile afile)
68     {
69         jfiles.put( name, afile );
70         setModdate();
71     }
72
73     /**
74      * Get a JFile in this Directory, by name .
75
76      * @param filename the name of the JFile to find.
77      * @return the JFile found.
78      */
79
80     public JFile retrieveJFile( String filename )
81     {
82         JFile afile = (JFile)jfiles.get( filename );
83         return afile;
84     }
85
86     /**
87      * Remove a JFile in this Directory, by name .
88
89      * @param filename the name of the JFile to remove
90      */
91
92     public void removeJFile( String filename )
93     {
94         jfiles.remove( filename );
95     }
96
97     /**
98      * Get the contents of this Directory as an array of
99      * the file names, each of which is a String.
100
101      * @return the array of names.
102      */
103
104     public String[] getFileNames()
105     {
106         return (String[])jfiles.keySet().toArray( new String[0] );
107     }
108 }

```

```

1 // fo1/10/juno/TextFile.java
2 //
3 //
4 // Copyright 2003 Ethan Bolker and Bill Campbell
5
6 /**
7  * A TextFile is a JFile that holds text.
8  *
9  * @version 10
10 */
11
12 public class TextFile extends JFile
13 {
14     private String contents; // The text itself
15
16     /**
17      * Construct a TextFile with initial contents.
18      *
19      * @param name the name for this TextFile (in its parent Directory).
20      * @param creator the owner of this new TextFile
21      * @param parent the Directory in which this TextFile lives.
22      * @param initialContents the initial text
23      */
24
25     public TextFile( String name, User creator, Directory parent,
26                     String initialContents )
27     {
28         super( name, creator, parent );
29         setContents( initialContents );
30     }
31
32     /**
33      * Construct an empty TextFile.
34      *
35      * @param name the name for this TextFile (in its parent Directory).
36      * @param creator the owner of this new TextFile
37      * @param parent the Directory in which this TextFile lives
38      */
39
40     TextFile( String name, User creator, Directory parent )
41     {
42         this( name, creator, parent, "" );
43     }
44
45     /**
46      * The size of a text file is the number of characters stored.
47      *
48      * @return the file's size.
49      */
50
51     public int getSize()
52     {
53         return contents.length();
54     }
55
56     /**

```

```

57      * Suffix used for printing text file names is "".
58      *
59      * @return an empty suffix (for TextFiles).
60      */
61
62     public String getSuffix()
63     {
64         return "";
65     }
66
67     /**
68      * Replace the contents of the file.
69      *
70      * @param contents the new contents.
71      */
72
73     public void setContents( String contents )
74     {
75         this.contents = contents;
76         setModDate();
77     }
78
79     /**
80      * The contents of a text file.
81      *
82      * @return String contents of the file.
83      */
84
85     public String getContents()
86     {
87         return contents;
88     }
89
90     /**
91      * Append text to the end of the file.
92      *
93      * @param text the text to be appended.
94      */
95
96     public void append( String text )
97     {
98         setContents( contents + text );
99     }
100
101     /**
102      * Append a new line of text to the end of the file.
103      *
104      * @param text the text to be appended.
105      */
106
107     public void appendLine( String text )
108     {
109         this.setContents( contents + '\n' + text );
110     }
111
112     }

```

```

1 // fo1/10/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, password,
8  * a home directory, and a real name.
9  * name.
10 *
11 * @version 10
12 */
13
14 public class User
15 implements java.io.Serializable
16 {
17     private String name; // The user's login name
18     private String password; // The user's login password.
19     private Directory home; // her home Directory
20     private String realName; // her real name
21
22     /**
23      * Construct a new User.
24      *
25      * @param name the User's login name.
26      * @param password the user's login password.
27      * @param home her home Directory.
28      * @param realName her real name.
29      */
30
31     public User( String name, String password,
32                 Directory home, String realName )
33     {
34         this.name = name;
35         this.password = password;
36         this.home = home;
37         this.realName = realName;
38     }
39
40     /**
41      * Confirm password. Throw a JunoException on failure.
42      *
43      * @param guess the string to test against the password.
44      *
45      * @exception JunoException
46      *             if password fails to match
47      */
48
49     public void matchPassword( String guess ) throws JunoException
50     {
51         if ( !guess.equals( password ) ) {
52             throw new JunoException( "bad password" );
53         }
54     }
55
56     /**

```

```

57      * Get the User's login name.
58      *
59      * @return the name.
60      */
61
62     public String getName()
63     {
64         return name;
65     }
66
67     /**
68      * Convert the User to a String.
69      * The String representation for a User is her
70      * login name.
71      *
72      * @return the User's name.
73      */
74
75     public String toString()
76     {
77         return getName();
78     }
79
80     /**
81      * Get the User's home Directory.
82      *
83      * @return the home Directory.
84      */
85
86     public Directory getHome()
87     {
88         return home;
89     }
90
91     /**
92      * Get the user's real name.
93      *
94      * @return the real name.
95      */
96
97     public String getRealName()
98     {
99         return realName;
100     }
101 }

```



```
1 // fo1/10/juno/JunoException.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A general Juno Exception.
8  *
9  * @version 10
10 */
11
12 public class JunoException extends Exception
13 {
14     /**
15      * The default (no argument) constructor.
16      */
17
18     public JunoException()
19     {
20     }
21
22     /**
23      * A general Juno exception holding a String message.
24      *
25      * @param message the message.
26      */
27
28     public JunoException( String message )
29     {
30         // Exception (actually Throwable, Exceptions's superclass)
31         // can remember the String passed its constructor.
32
33         super( message );
34     }
35
36     // Note, to get the message stored in a JunoException
37     // we can just use the (inherited) methods getMessage(),
38     // and toString().
39 }
```

```
1 // foj/10/Juno/BadShellCommandException.java
2 //
3 //
4 // Copyright 2003 Ethan Bolker and Bill Campbell
5
6 /**
7  * The Exception generated when a ShellCommand is misused.
8  *
9  * @version 1.0
10 */
11
12 class BadShellCommandException extends JunoException
13 {
14     private ShellCommand command;
15
16     /**
17     * Construct a new BadShellCommandException
18     * containing the badly used command.
19     *
20     * @param the ShellCommand being misused.
21     */
22
23     public BadShellCommandException( ShellCommand command )
24     {
25         this.command = command;
26     }
27
28     /**
29     * Get the command.
30     */
31
32     public ShellCommand getCommand()
33     {
34         return command;
35     }
36 }
```

```
1 // fo1/10/juno/ExitShellException.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * Exception raised for exiting a shell.
8  *
9  * @version 10
10 */
11
12 public class ExitShellException extends JunoException
13 {
14 }
```

```
1 // foj/10/Juno/ShellCommandNotFoundException.java (version 10)
2 //
3 //
4 // Copyright 1997-2001 Ethan Bolker and Bill Campbell
5
6 /**
7  * The Exception when a ShellCommand isn't found.
8  */
9
10 class ShellCommandNotFoundException extends JunoException
11 {
12     /**
13      * Create a ShellCommandNotFoundException.
14      */
15
16     public ShellCommandNotFoundException()
17     {
18     }
19
20     /**
21      * Create a ShellCommandNotFoundException with
22      * a message reporting the command tried.
23      */
24
25     public ShellCommandNotFoundException(String commandName )
26     {
27         super( "ShellCommand " + commandName + " not found" );
28     }
29 }
```

```
1 // fo1/10/juno/JFileNotFoundException.java (version 10)
2 //
3 //
4 // Copyright 1997-2001 Ethan Bolker and Bill Campbell
5
6 /**
7  * The Exception thrown when a JFile isn't found
8  *
9  * @version 10
10 */
11
12 class JFileNotFoundException extends JunoException
13 {
14     String jfilename;
15
16     /**
17      * Construct a new JFileNotFoundException
18      *
19      * @param jfilename the file sought
20      */
21
22     public JFileNotFoundException( String jfilename )
23     {
24         super( "JFile " + jfilename + " not found." );
25         this.jfilename = jfilename;
26     }
27
28     /**
29      * Get the name of the file that wasn't there.
30      *
31      * @return the file name
32      */
33
34     public String getJfilename()
35     {
36         return jfilename;
37     }
38 }
```

```

1 // fo1/10/juno/GUILoginConsole.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import javax.swing.*;
7 import javax.swing.event.*;
8 import java.awt.*;
9 import java.awt.event.*;
10
11 /**
12  * The graphical user interface to Juno.
13  */
14
15 public class GUILoginConsole extends JFrame
16 implements OutputInterface
17 {
18     private static final int FIELDWIDTH = 30;
19     private static final int FIELDHEIGHT = 5;
20
21     private final Juno junoSystem;
22     private WindowCloser closeMe; // to shut down Juno
23
24     private String title; // title for the windows
25
26     // The interpreter interprets one-line commands.
27     private InterpreterInterface interpreter;
28     private boolean echoInput;
29
30     // All output goes to messages.
31     private JTextArea messages;
32
33     /**
34      * Construct a GUI console for Juno.
35      *
36      * @param title the title for this window.
37      * @param junoSystem the Juno system for which this is a GUI
38      * @param interpreter the object to which to send user input.
39      * @param echoInput true when input echoes to this console.
40      */
41
42     public GUILoginConsole( String title, Juno junoSystem,
43                           InterpreterInterface interpreter,
44                           boolean echoInput)
45     {
46         super( title );
47         this.title = title;
48         this.junoSystem = junoSystem;
49         this.interpreter = interpreter;
50         this.echoInput = echoInput;
51         this.closeMe = new WindowCloser( junoSystem );
52
53         // Set up the look and feel;
54         // Everything is placed on a panel (using BorderLayout)
55         JPanel panel = new JPanel();

```

```

57     panel.setLayout( new BorderLayout() );
58
59     // First a tabbed pane, with two tabs:
60     // one for login, one for registration
61
62     JTabbedPane tabs = new JTabbedPane();
63     tabs.addTab( "Login", null,
64               new LoginPane( interpreter, echoInput, closeMe ) );
65     tabs.addTab( "Register", null,
66               new RegisterPane( interpreter, echoInput ) );
67     tabs.setSelectedIndex( 0 ); // Login selected by default
68     panel.add( tabs, BorderLayout.NORTH );
69
70     // and the output messages area.
71     panel.add( new JLabel( "Messages:" ), BorderLayout.CENTER );
72     messages = new JTextArea( FIELDHEIGHT, FIELDWIDTH );
73     panel.add( messages, BorderLayout.SOUTH );
74
75     // Add the panel to this JFrame
76     this.getContentPane().add( panel );
77
78     // Closing this window
79     this.setDefaultCloseOperation( JFrame.DO_NOTHING_ON_CLOSE );
80     this.addWindowListener( closeMe );
81
82     // Size and display this JFrame
83     pack();
84     show();
85
86     // Implementing the OutputInterface. Everything goes to the
87     // single message area.
88
89     /**
90      * Write a String followed by a newline
91      * to message area.
92      *
93      * @param str - the string to write
94      */
95
96     public void println( String str )
97     {
98         messages.append( str + "\n" );
99     }
100
101     /**
102      * Write a String followed by a newline
103      * to message area.
104      *
105      * @param str - the String to write
106      */
107
108     public void errPrintln( String str )
109     {
110         println( str );
111     }
112

```

```

113
114 /**
115  * Query what kind of console this is.
116  *
117  * @return true if and only if echoing input.
118  */
119
120 public boolean isEchoInput()
121 {
122     return echoInput;
123 }
124
125 /**
126  * Query what kind of console this is.
127  *
128  * @return true if and only if GUI
129  */
130
131 public boolean isGUI()
132 {
133     return true;
134 }
135
136 /**
137  * Query what kind of console this is.
138  *
139  * @return true if and only if remote
140  */
141
142 public boolean isRemote()
143 {
144     return false;
145 }
146
147 // The Login pane is specified in a private inner class,
148 // visible only here.
149
150 private class LoginPane extends JPanel
151 {
152     // The login pane has two text fields and two buttons.
153     private JTextField nameField;
154     private JTextField passwordField;
155
156     private JButton ok;
157     private JButton exit;
158
159     private WindowCloser closeMe; // to shut down Juno
160     // Construct the login pane and set up its listeners.
161
162     public LoginPane( InterpreterInterface interpreter,
163                     boolean echoInput, WindowCloser closeMe )
164     {
165         super();
166         this.closeMe = closeMe;
167     }
168     // Set up the look and feel.

```

```

169
170 // Everything will go into a vertical Box, a container
171 // whose contents are laid out using BoxLayout
172 Box box = Box.createVerticalBox();
173
174 // First a panel, containing the two text fields
175
176 JPanel p = new JPanel();
177 p.setLayout( new GridLayout( 4, 1 ) );
178
179 p.add( new JLabel( "Login:" ) );
180 nameField = new JTextField( FIELDWIDTH );
181 p.add( nameField );
182
183 p.add( new JLabel( "Password:" ) );
184 passwordField = new JPasswordField( FIELDWIDTH );
185 p.add( passwordField );
186
187 box.add( p );
188 box.add( Box.createVerticalStrut( 15 ) );
189
190 // Then a horizontal Box containing the two buttons
191 Box row = Box.createHorizontalBox();
192 row.add( Box.createGlue() );
193
194 ok = new JButton( "OK" );
195 row.add( ok );
196 row.add( Box.createGlue() );
197
198 exit = new JButton( "Exit" );
199 row.add( exit );
200 row.add( Box.createGlue() );
201 box.add( row );
202 box.add( Box.createVerticalStrut( 15 ) );
203
204 this.setLayout( new BorderLayout() );
205 this.add( box, BorderLayout.CENTER );
206
207 // Set up the listeners (the semantics)
208
209 ok.addActionListener( new LoginProcessor() );
210 exit.addActionListener( closeMe ); // shuts down Juno
211
212 }
213
214 // An inner class for the semantics
215 // when the user clicks OK.
216
217 private class LoginProcessor implements ActionListener
218 {
219     public void actionPerformed( ActionEvent e )
220     {
221         String str = nameField.getText() + " " +
222             passwordField.getText();
223         passwordField.setText( "" );
224         messages.setText( str + "\n" ); // for debugging

```

```

225         interpreter.interpret( str );
226     }
227 }
228 }
229 // The Register pane is specified in a private inner class,
230 // visible only here.
231
232 private class RegisterPane extends JPanel
233 {
234     // The register pane has four textfields and two buttons.
235     private JTextField chosenName;
236     private JTextField fullName;
237     private JTextField password1;
238     private JTextField password2;
239
240     private JButton register;
241     private JButton clear;
242
243     public RegisterPane( InterpreterInterface interpreter,
244                         boolean echoInput )
245     {
246         super();
247
248         // Define the look and feel
249         // Everything goes into a vertical Box
250         Box box = Box.createVerticalBox();
251
252         // First a panel containing the text fields
253         JPanel p = new JPanel();
254         p.setLayout( new GridLayout( 0 , 1 ) );
255
256         p.add( new JLabel( "Choose login name:" ) );
257         chosenName = new JTextField( FIELDWIDTH );
258         p.add( chosenName );
259
260         p.add( new JLabel( "Give full name:" ) );
261         fullName = new JTextField( FIELDWIDTH );
262         p.add( fullName );
263
264         p.add( new JLabel( "Choose password:" ) );
265         password1 = new JTextField( FIELDWIDTH );
266         p.add( password1 );
267
268         p.add( new JLabel( "Retype password:" ) );
269         password2 = new JTextField( FIELDWIDTH );
270         p.add( password2 );
271
272         box.add( p );
273
274         box.add( Box.createVerticalStrut( 15 ) );
275
276         // Then a horizontal Box containing the buttons
277         Box row = Box.createHorizontalBox();
278         row.add( Box.createGlue() );
279
280

```

```

281         register = new JButton( "Register" );
282         row.add( register );
283         row.add( Box.createGlue() );
284         clear = new JButton( "Clear" );
285         row.add( clear );
286         row.add( Box.createGlue() );
287         box.add( row );
288         box.add( Box.createVerticalStrut( 15 ) );
289
290         this.setLayout( new BorderLayout() );
291         this.add( box, BorderLayout.CENTER );
292
293         // Set up the listeners (the semantics)
294         register.addActionListener( new Registration() );
295         clear.addActionListener( new Cleanser() );
296     }
297
298     // An inner class for the semantics when the user
299     // clicks Register.
300     private class Registration implements ActionListener
301     {
302         public void actionPerformed( ActionEvent e )
303         {
304             if ( password1.getText().trim().equals(
305                 password2.getText().trim() ) ) {
306                 String str = "register " +
307                     chosenName.getText() + " " +
308                     password1.getText() + " " +
309                     fullName.getText();
310                 chosenName.setText("");
311                 fullName.setText("");
312                 messages.setText( str + '\n' ); // for debugging
313                 interpreter.interpret( str );
314             }
315             else {
316                 messages.setText(
317                     "Sorry, passwords don't match.\n" );
318             }
319         }
320     }
321
322     // An inner class for the semantics when the user
323     // clicks Clear.
324     private class Cleanser implements ActionListener {
325         public void actionPerformed( ActionEvent e ) {
326             chosenName.setText("");
327             fullName.setText("");
328             password1.setText("");
329             password2.setText("");
330         }
331     }
332 }
333
334
335
336

```



```
337     }
338
339     // A WindowCloser instance handles close events generated
340     // by the underlying window system with its windowClosing
341     // method, and close events from buttons or other user
342     // components with its actionPerformed method.
343     //
344     // The action is to shut down Juno.
345
346     private static class WindowCloser extends WindowAdapter
347     implements ActionListener
348     {
349         Juno system;
350
351         public WindowCloser( Juno system )
352         {
353             this.system = system;
354         }
355
356         public void windowClosing (WindowEvent e)
357         {
358             this.actionPerformed( null );
359         }
360
361         public void actionPerformed(ActionEvent e)
362         {
363             if (system != null) {
364                 system.shutdown();
365             }
366             System.exit(0);
367         }
368     }
369
370     /**
371     * main() in GUILoginConsole class for
372     * unit testing during development.
373     */
374
375     public static void main( String[] args )
376     {
377         new GUILoginConsole( "GUITest", null, null, true ).show();
378     }
379 }
380
```

```

1 // fo1/10/juno/GUIShellConsole.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import javax.swing.*;
7 import java.awt.*;
8 import java.awt.event.*;
9 import java.util.*;
10
11 /**
12  * The GUI to the Juno system Shell.
13  */
14
15 public class GUIShellConsole extends JFrame
16 implements OutputInterface
17 {
18     private static final int FIELDWIDTH = 50;
19     private static final int FIELDHEIGHT = 10;
20
21     // the components on the window
22
23     private JLabel promptLabel = new JLabel();
24     private JTextField commandLine = new JTextField( FIELDWIDTH );
25     private JButton doIt = new JButton( "Do It" );
26     private JButton logout = new JButton( "Logout" );
27     private JTextArea stdout =
28         new JTextArea( FIELDHEIGHT, FIELDWIDTH );
29     private JTextArea stderr =
30         new JTextArea( FIELDHEIGHT/2, FIELDWIDTH );
31
32     private Shell sh; // for interpreting shell commands
33     private WindowCloser closer; // for logging out.
34
35     private boolean echoInput;
36
37     /**
38      * Construct a GUI console for a shell.
39      *
40      * @param title the title to display in the frame.
41      * @param sh the shell to interpret commands.
42      * @param echoInput is input to be echoed?
43      */
44
45     public GUIShellConsole( String title,
46                             Shell sh,
47                             boolean echoInput )
48     {
49         this.sh = sh;
50         this.echoInput = echoInput;
51
52         setTitle( title );
53         setPrompt( sh.getPrompt() );
54
55         // set up console's look and feel
56

```

```

57         JPanel outerPanel = new JPanel();
58         outerPanel.setLayout( new BorderLayout() );
59
60         Box box = Box.createVerticalBox();
61
62         JPanel commandPanel = new JPanel();
63         commandPanel.setLayout( new BorderLayout() );
64         commandPanel.add( promptLabel, BorderLayout.NORTH );
65         commandPanel.add( commandLine, BorderLayout.CENTER );
66         box.add( commandPanel );
67         box.add( Box.createVerticalStrut( 10 ) );
68
69         Box buttons = Box.createHorizontalBox();
70         buttons.add( Box.createGlue() );
71         buttons.add( doIt );
72         buttons.add( Box.createGlue() );
73         buttons.add( logout );
74         buttons.add( Box.createGlue() );
75         box.add( buttons );
76         box.add( Box.createVerticalStrut( 10 ) );
77
78         JPanel stdoutPanel = new JPanel();
79         stdoutPanel.setLayout( new BorderLayout() );
80         stdoutPanel.add( new JLabel( "Standard output:" ),
81                         BorderLayout.NORTH );
82
83         stdoutPanel.add( new JScrollPane( stdout ),
84                         BorderLayout.CENTER );
85
86         box.add( stdoutPanel );
87         box.add( Box.createVerticalStrut( 10 ) );
88         stdout.setEditable( false );
89
90         JPanel stderrPanel = new JPanel();
91         stderrPanel.setLayout( new BorderLayout() );
92         stderrPanel.add( new JLabel( "Error output:" ),
93                         BorderLayout.NORTH );
94         stderrPanel.add( new JScrollPane( stderr ),
95                         BorderLayout.CENTER );
96         box.add( stderrPanel );
97         box.add( Box.createVerticalStrut( 10 ) );
98         stderr.setEditable( false );
99
100        outerPanel.add( box, BorderLayout.CENTER );
101        this.getContentPane().add( outerPanel, BorderLayout.CENTER );
102
103        // Install menus and tool bar.
104
105        JMenuBar menuBar = new JMenuBar();
106        JMenu commandMenu = new JMenu( "Command" );
107        JMenu helpMenu = new JMenu( "Help" );
108
109        JToolBar toolBar = new JToolBar();
110
111        // Create menu items and tool buttons for each shell command
112

```

```

113 ShellCommandTable table = sh.getSystem().getCommandTable();
114 String [] commandNames = table.getCommandNames();
115 for ( int i = 0; i < commandNames.length; i++ ) {
116     String commandName = commandNames[i];
117     ShellCommand command =
118         table.lookup( commandName );
119     CommandMenuAction commandAction =
120         new CommandMenuAction( commandName,
121             command.getArgString() );
122     HelpMenuAction helpAction =
123         new HelpMenuAction( commandName,
124             command.getArgString(),
125             command.getHelpString() );
126     JMenuItem item1 = commandMenu.add( commandAction );
127     JMenuItem item2 = helpMenu.add( helpAction );
128     JButton button = toolbar.add( commandAction );
129     JButton button.setText( command.getHelpString() );
130 }
131 this.setMenuBar( menuBar );
132 this.getContentPane().add( toolbar,
133     BorderLayout.NORTH );
134 menuBar.add( commandMenu );
135 menuBar.add( helpMenu );
136 pack();
137 show();
138 // add listener to the Do It button
139 doIt.addActionListener( new Interpreter() );
140 // add listener to the Logout button and window closer
141 closeMe = new WindowCloser( this );
142 logout.addActionListener( closeMe );
143 this.addWindowListener( closeMe );
144 }
145 // Set the GUI prompt
146 private void setPrompt( String prompt )
147 {
148     this.promptLabel.setText( prompt );
149 }
150 // Implementing the OutputInterface.
151 // Everything goes to the single message area.
152 public void println( String str )
153 {
154     stdout.append( str + "\n" );
155 }

```

```

169     }
170     public void errprintln( String str )
171     {
172         stderr.append( str + "\n" );
173     }
174     public boolean isGUI()
175     {
176         return true;
177     }
178     public boolean isRemote()
179     {
180         return false;
181     }
182     public boolean isEchoInput()
183     {
184         return echoInput;
185     }
186     // An inner class for the semantics when the user submits
187     // a ShellCommand for execution.
188     private class Interpreter
189     implements ActionListener
190     {
191         public void actionPerformed( ActionEvent e )
192         {
193             String str = commandLine.getText();
194             stdout.append( sh.getPrompt() + str + '\n' );
195             if ( sh.interpret( str ) ) {
196                 setPrompt( sh.getPrompt() );
197             }
198             else {
199                 closeMe.actionPerformed( null );
200             }
201         }
202     }
203     private class CommandMenuAction extends AbstractAction
204     {
205         private String argString;
206         private String helpString;
207         public CommandMenuAction( String text, String argString )
208         {
209             super( text );
210             this.argString = argString;
211         }
212         public void actionPerformed( ActionEvent e )
213         {
214             commandLine.setText( getValue( Action.NAME ) +
215                 " " + argString );
216         }
217     }

```

```
225     }
226   }
227
228   private class HelpMenuAction extends AbstractAction
229   {
230     private String argString;
231     private String helpString;
232
233     public HelpMenuAction( String text, String argString,
234                           String helpString )
235     {
236       super( text );
237       this. argString = argString;
238       this.helpString = helpString;
239     }
240
241     public void actionPerformed( ActionEvent e )
242     {
243       stdout.append( getValue( Action.NAME ) + " : " +
244                     helpString );
245     }
246   }
247
248   // A WindowCloser instance handles close events generated
249   // by the underlying window system with its windowClosing
250   // method, and close events from buttons or other user
251   // components with its actionPerformed method.
252   //
253   // The action is to logout and dispose of this window.
254
255   private static class WindowCloser extends WindowAdapter
256   implements ActionListener
257   {
258     Frame myFrame;
259
260     public WindowCloser( Frame frame ) {
261       myFrame = frame;
262     }
263
264     public void windowClosing (WindowEvent e)
265     {
266       this.actionPerformed( null );
267     }
268
269     public void actionPerformed(ActionEvent e)
270     {
271       myFrame.dispose();
272     }
273   }
274 }
```

```
1 // foj/10/InterpreterInterface.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * Jumo needs an interpreter to process the user's response to
8  * the login: prompt (or what she enters on a GUILoginConsole).
9
10 * Each Shell needs an interpreter for shell command lines,
11 * whether entered with a GUI or a CLI.
12 *
13 * @version 10
14 */
15
16 public interface InterpreterInterface
17 {
18     /**
19      * Interpret a command line String.
20      *
21      * @param str the String to interpret
22      * @return true, unless str tells you there's nothing to follow
23      */
24     public boolean interpret( String str );
25 }
26
```

```
1 // fo1/10/juno/InputInterface.java
2 //
3 //
4 // Copyright 2003 Ethan Bolker and Bill Campbell
5
6 /**
7  * Juno consoles use the same abstract method
8  * for input, so it is specified here.
9  */
10
11 public interface InputInterface
12 {
13     /**
14      * Read a line (terminated by a newline).
15      *
16      * @param promptString output string to prompt for input
17      * @return the string (without the newline character)
18      */
19     public String readLine( String promptString );
20 }
21
22
```

```
1 // fo1/10/juno/OutputInterface.java
2 //
3 //
4 // Copyright 2003 Ethan Bolker and Bill Campbell
5
6 /**
7  * All Juno consoles use the same abstract methods
8  * for output, so they are specified here.
9  */
10
11 public interface OutputInterface
12 {
13     /**
14      * Write a String followed by a newline
15      * to console output location.
16      *
17      * @param str - the string to write
18      */
19
20     public void println(String str );
21
22     /**
23      * Write a String followed by a newline
24      * to console error output location.
25      *
26      * @param str - the String to write
27      */
28
29     public void errPrintln( String str );
30
31     /**
32      * Query what kind of console this is.
33      *
34      * @return true if and only if echoing input.
35      */
36
37     public boolean isEchoInput();
38
39     /**
40      * Query what kind of console this is.
41      *
42      * @return true if and only if GUI
43      */
44
45     public boolean isGUI();
46
47     /**
48      * Query what kind of console this is.
49      *
50      * @return true if and only if remote
51      */
52
53     public boolean isRemote();
54 }
55
```

```

1 // fo1/10/juno/JunoTerminal.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A Command line interface terminal for Juno.
8  *
9  * @version 1.0
10 */
11
12 public class JunoTerminal
13 implements InputInterface, OutputInterface
14 {
15     private Terminal terminal; // the delegate terminal
16     private boolean echo; // are we echoing input?
17
18     /**
19      * Construct a JunoTerminal
20      *
21      * Allows for input echo, when, for example, input is redirected
22      * from a file.
23      *
24      * @param echo whether or not input should be echoed.
25      */
26
27     public JunoTerminal( boolean echo )
28     {
29         this.echo = echo;
30         terminal = new Terminal( echo );
31     }
32
33     // Implement InputInterface
34
35     /**
36      * Read a line (terminated by a newline).
37      *
38      * @param promptString output string to prompt for input
39      * @return the string (without the newline character)
40      */
41
42     public String readline( String promptString )
43     {
44         return terminal.readline( promptString );
45     }
46
47     // Implement OutputInterface
48
49     /**
50      * Write a String followed by a newline
51      * to console output location.
52      *
53      * @param str - the string to write
54      */
55
56     public void println( String str )

```

```

57     {
58         terminal.println( str );
59     }
60
61     /**
62      * Write a String followed by a newline
63      * to console error output location.
64      *
65      * @param str - the String to write
66      */
67
68     public void errPrintln( String str )
69     {
70         terminal.errPrintln( str );
71     }
72
73     /**
74      * Query what kind of console this is.
75      *
76      * @return true if and only if echoing input.
77      */
78
79     public boolean isEchoInput()
80     {
81         return echo;
82     }
83
84     /**
85      * Query what kind of console this is.
86      *
87      * @return false, since it is not a GUI
88      */
89
90     public boolean isGUI()
91     {
92         return false;
93     }
94
95     /**
96      * Query what kind of console this is.
97      *
98      * @return false, since it is not remote.
99      */
100
101     public boolean isRemote()
102     {
103         return false;
104     }
105 }

```



```

1 // fo1/10/juno/RemoteConsole.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.io.*;
7 import java.net.*;
8 import java.util.*;
9 import java.text.*;
10
11 /**
12  * A remote console listens on a port for a remote login to
13  * a running Juno system server.
14  *
15  * @version 1.0
16  */
17
18 public class RemoteConsole extends Thread
19 implements OutputInterface, InputInterface
20 {
21     // default just logs connection start and end
22     // change to true to log all i/o
23     private static boolean logall = false;
24
25     private Juno system;
26     private boolean echo;
27     private InterpreterInterface interpreter;
28
29     private Socket clientSocket;
30     private BufferedReader in;
31     private PrintWriter out;
32     private int sessionCount = 0;
33
34     private PrintWriter junolog;
35
36     /**
37      * Construct a remote console to listen for users trying
38      * to connect to Juno.
39      *
40      * Called from Juno main.
41      *
42      * @param system the Juno system setting up this console.
43      * @param echo whether or not input should be echoed.
44      * @param port the port on which to listen for requests.
45      */
46
47     public RemoteConsole( Juno system, boolean echo, int port )
48     {
49         this.echo = echo;
50         Date now = new Date();
51         junolog = openlog(now);
52         log("*** Juno server started " + now + "\n");
53         try {
54             ServerSocket ss = new ServerSocket(port);
55             while (true) {
56                 clientSocket = ss.accept();

```

```

57         new RemoteConsole( system, echo, clientSocket,
58                             junolog, ++sessionCount).start();
59     }
60 }
61 catch (IOException e) {
62     System.out.println("Remote login not supported");
63     System.exit(0);
64 }
65 finally {
66     system.shutdown();
67 }
68
69 /**
70  * Construct a remote console for a single remote user.
71  *
72  * @param system the Juno system to which the user is connecting.
73  * @param echo whether or not input should be echoed.
74  * @param clientSocket the socket for the user's connection
75  * @param junolog track all user i/o
76  * @param sessionCount this session's number
77  */
78
79 public RemoteConsole( Juno system, boolean echo, Socket clientSocket,
80                       PrintWriter junolog, int sessionCount )
81 {
82     this.system = system;
83     this.echo = echo;
84     this.clientSocket = clientSocket;
85     this.junolog = junolog;
86     this.sessionCount = sessionCount;
87 }
88
89 /**
90  * Action when the thread for this session starts.
91  */
92
93 public void run()
94 {
95     log("*** " + sessionCount + ", " +
96         clientSocket.getInetAddress() + ", " +
97         new Date());
98     try {
99         setUpConnection();
100        String s = this.readLine
101            ("Please sign the guest book (name, email): ");
102        this.println("Thanks, " + s);
103        if (!logall) {
104            log("guest book: " + s);
105        }
106        new LoginInterpreter( system, this ).login();
107        clientSocket.close();
108    }
109    catch (IOException e) {
110        log("*** Error " + e);
111    }
112 }

```

```

113     log("*** end session " + sessionCount);
114     }
115     /**
116     * Create the readers and writers for the socket
117     * for this session.
118     */
119     private void setUpConnection()
120     throws IOException
121     {
122         in = new BufferedReader(
123             new InputStreamReader(clientSocket.getInputStream()));
124         out = new PrintWriter(
125             new OutputStreamWriter(clientSocket.getOutputStream()));
126     }
127     // implement the InputInterface
128     /**
129     * Read a line (terminated by a newline) from console socket.
130     */
131     * Log the input line before returning it if required.
132     *
133     * @param promptString output string to prompt for input
134     * @return the string (without the newline character)
135     */
136     public String readline( String promptString )
137     {
138         String s = "";
139         this.print(promptString);
140         out.flush();
141         try {
142             s = in.readLine();
143             if (logall) {
144                 log("> " + s);
145             }
146             if (echo) {
147                 out.println(s);
148             }
149             catch (IOException e) {
150                 String msg = "IO error reading from remote console";
151                 System.out.println(msg);
152                 out.println(msg);
153             }
154             return s;
155         }
156     }
157     /**
158     * Write a String to console socket.
159     *
160     * Log the output if required.
161     *
162     * @param str - the string to write
163     */

```

```

169     */
170     public void print( String str )
171     {
172         out.print( str );
173         out.flush();
174         if (logall) {
175             log("< " + str + "\\");
176         }
177     }
178     // implement the OutputInterface
179     /**
180     * Write a String followed by a newline
181     * to console socket.
182     *
183     * Log the output if required.
184     *
185     * @param str - the string to write
186     */
187     public void println( String str )
188     {
189         out.println( str + '\r' );
190         out.flush();
191         if (logall) {
192             log("< " + str);
193         }
194     }
195     /**
196     * Write a String followed by a newline
197     * to console error output location. That's
198     * just the socket.
199     *
200     * @param str - the String to write
201     */
202     public void errPrintln(String str )
203     {
204         println( str );
205     }
206     /**
207     * Query what kind of console this is.
208     *
209     * @return false since it's not a GUI.
210     */
211     public boolean isGUI()
212     {
213         return false;
214     }
215     }
216     /**
217     */
218     }
219     }
220     }
221     }
222     }
223     }
224     }

```

```
225     * Query what kind of console this is.
226     * @return true since it is remote.
227     */
228     public boolean isRemote()
229     {
230         return true;
231     }
232 }
233
234 /**
235  * Query what kind of console this is.
236  * @return true if and only if echoing input.
237  */
238     public boolean isEchoInput()
239     {
240         return echo;
241     }
242 }
243
244 /**
245  * Log a String.
246  * @param str the String to log.
247  */
248     private void log(String str)
249     {
250         junolog.println(sessionCount + ": " + str);
251         junolog.flush();
252     }
253 }
254
255 /**
256  * Open a log for this console.
257  * @param now the current Date.
258  */
259     private PrintWriter openlog(Date now)
260     {
261         PrintWriter out = null;
262         SimpleDateFormat formatter
263             = new SimpleDateFormat ("MMM.dd:hh:mm:ss");
264         String dateString = formatter.format(now);
265         String filename = "log-" + dateString;
266         try { out = new PrintWriter(
267             new BufferedWriter(
268                 new FileWriter(filename)));
269         }
270         catch (Exception e) {
271             out = new PrintWriter(new FileWriter(FileDescriptor.out));
272         }
273         return out;
274     }
275 }
276
277
278
279
280 }
```