

```

1 // fo1/7/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  * like 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 7
22  */
23
24 public class Juno
25 {
26     private final static String os      = "Juno";
27     private final static String version = "7";
28
29     private String  hostname; // host machine name
30     private Map    users;    // lookup table for Users
31     private Terminal console; // for input and output
32
33     private Directory slash; // root of JFile system
34     private Directory userHomes; // for home directories
35
36     private ShellCommandTable commandTable; // shell commands
37
38     /**
39      * Construct a Juno (operating system) object.
40      *
41      * @param hostname the name of the host on which it's running.
42      * @param echoInput should all input be echoed as output?
43      */
44
45     public Juno( String hostname, boolean echoInput )
46     {
47         // initialize the Juno environment ...
48
49         this.hostname = hostname;
50         console       = new Terminal( echoInput );
51         users         = new TreeMap(); // for registered Users
52         commandTable = new ShellCommandTable(); // for shell commands
53
54         // the file system
55
56         slash = new Directory( "", null, null );

```

```

57     User root = new User( "root", slash, "Rick Martin" );
58     users.put( "root", root );
59     slash.setOwner( root );
60     userHomes = new Directory( "users", root, slash );
61
62     // create, then start a command line login interpreter
63     LoginInterpreter interpreter
64     = new LoginInterpreter( this, console );
65     interpreter.CLIlogin();
66
67 }
68
69 /**
70  * The name of the host computer on which this system
71  * is running.
72  *
73  * @return the host computer name.
74  */
75
76     public String getHostName()
77     {
78         return hostname;
79     }
80
81     /**
82      * The name of this operating system.
83      *
84      * @return the operating system name.
85      */
86
87     public String getOS()
88     {
89         return os;
90     }
91
92     /**
93      * The version number for this system.
94      *
95      * @return the version number.
96      */
97
98     public String getVersion()
99     {
100        return version;
101    }
102
103    /**
104     * The directory containing all user homes for this system.
105     *
106     * @return the directory containing user homes.
107     */
108
109     public Directory getUserHomes()
110     {
111         return userHomes;
112     }

```

```

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

```

```

1 // foj/7/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 7
14 */
15
16 public class LoginInterpreter
17 {
18     private static final String LOGIN_COMMANDS =
19         "help, register, <username>, exit";
20
21     private Juno    system; // the Juno object
22     private Terminal console; // for i/o
23
24     /**
25      * Construct a new LoginInterpreter for interpreting
26      * login commands.
27      *
28      * @param system the system creating this interpreter.
29      * @param console the Terminal used for input and output.
30      */
31
32     public LoginInterpreter( Juno system, Terminal console )
33     {
34         this.system = system;
35         this.console = console;
36     }
37
38     /**
39      * Set the console for this interpreter.  Used by the
40      * creator of this interpreter.
41      *
42      * @param console the Terminal to be used for input and output.
43      */
44
45     public void setConsole( Terminal console )
46     {
47         this.console = console;
48     }
49
50     /**
51      * Simulates behavior at login: prompt.
52      * CLI stands for "Command Line Interface".
53      */
54     public void CLILogin()
55     {
56

```

```

57         welcome();
58         boolean moreWork = true;
59         while( moreWork ) {
60             moreWork = interpret( console.readLine( "Juno login: " ) );
61         }
62     }
63
64     // Parse user's command line and dispatch appropriate
65     // semantic action.
66     //
67     // return true unless "exit" command or null inputline.
68
69     private boolean interpret( String inputline )
70     {
71         if (inputline == null) return false;
72         StringTokenizer st =
73             new StringTokenizer( inputline );
74         if (st.countTokens() == 0) {
75             return true; // skip blank line
76         }
77         String visitor = st.nextToken();
78         if (visitor.equals( "exit" )) {
79             return false;
80         }
81         if (visitor.equals( "register" )) {
82             register( st );
83         }
84         else if (visitor.equals( "help" )) {
85             help();
86         }
87         else {
88             User user = system.lookupUser( visitor );
89             new Shell( system, user, console );
90             return true;
91         }
92     }
93
94     // Register a new user, giving him or her a login name and a
95     // home directory on the system.
96     //
97     // StringTokenizer argument contains the new user's login name
98     // followed by full real name.
99
100     private void register( StringTokenizer st )
101     {
102         String userName = st.nextToken();
103         String realName = st.nextToken().trim();
104         Directory home = new Directory( userName, null,
105             system.getUserHomes() );
106         User user = system.createUser( userName, home, realName );
107         home.setOwner( user );
108     }
109
110     // Display a short welcoming message, and remind users of
111     // available commands.
112

```

```
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 private void help()
123 {
124     console.println( LOGIN_COMMANDS );
125     console.println("");
126 }
127
128 }
```

```

1 // fo1/7/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 7
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();
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) { // skip blank line
64         return true;
65     }
66     String commandName = st.nextToken();
67     ShellCommand commandObject =
68         system.getCommandTable().lookup( commandName );
69     if (commandObject == null ) {
70         console.errPrintln("Unknown command: " + commandName); // EEE
71         return true;
72     }
73     try {
74         commandObject.doit( st, this );
75     }
76     catch (ExitShellException e) {
77         return false;
78     }
79     catch (BadShellCommandException e) {
80         console.errPrintln( "Usage: " + commandName + " " +
81             e.getCommand().getArgString() ); // EEE
82     }
83     catch (JunoException e) {
84         console.errPrintln( e.getMessage() ); // EEE
85     }
86     catch (Exception e) {
87         console.errPrintln( "you should never get here" ); // EEE
88         console.errPrintln( e.toString() ); // EEE
89     }
90     return true;
91 }
92
93 // Strip characters from '#' to end of line, create and
94 // return a StringTokenizer for what's left.
95
96 private StringTokenizer stripComments( String line )
97 {
98     int commentIndex = line.indexOf('#');
99     if (commentIndex >= 0) {
100         line = line.substring(0,commentIndex);
101     }
102     return new StringTokenizer(line);
103 }
104
105 /**
106  * The prompt for the CLI.
107  *
108  * @return the prompt string.
109  */
110
111 public String getPrompt()
112 {

```

```
113     }
114     return system.getHostName() + "> ";
115 }
116 /**
117  * The User associated with this shell.
118  *
119  * @return the user.
120  */
121
122 public User getUser()
123 {
124     return user;
125 }
126
127 /**
128  * The current working directory for this shell.
129  *
130  * @return the current working directory.
131  */
132
133 public Directory getDot()
134 {
135     return dot;
136 }
137
138 /**
139  * Set the current working directory for this shell.
140  *
141  * @param dot the new working directory.
142  */
143
144 public void setDot(Directory dot)
145 {
146     this.dot = dot;
147 }
148
149 /**
150  * The console associated with this shell.
151  *
152  * @return the console.
153  */
154
155 public Terminal getConsole()
156 {
157     return console;
158 }
159
160 /**
161  * The Juno object associated with this Shell.
162  *
163  * @return the Juno instance that created this Shell.
164  */
165
166 public Juno getSystem()
167 {
168     return system;
169 }
```

```
169     }
170 }
```

```

1 // fo1/7/juno/ShellCommand.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5 import java.util.*;
6
7 /**
8  * Model those features common to all ShellCommands.
9  *
10 * Each concrete extension of this class provides a constructor
11 * and an implementation for method doit().
12 *
13 * @version 7
14 */
15
16 public abstract class ShellCommand
17 {
18     private String helpString; // documents the command
19     private String argString; // any args to the command
20
21     /**
22      * A constructor, always called (as super()) by the subclass.
23      * Used only for commands that have arguments.
24      *
25      * @param helpString a brief description of what the command does.
26      * @param argString a prototype illustrating the required arguments.
27      */
28
29     protected ShellCommand( String helpString, String argString )
30     {
31         this.argString = argString;
32         this.helpString = helpString;
33     }
34
35     /**
36      * A constructor for commands having no arguments.
37      *
38      * @param helpString a brief description of what the command does.
39      */
40
41     protected ShellCommand( String helpString )
42     {
43         this( helpString, "" );
44     }
45
46     /**
47      * Execute the command.
48      *
49      * @param args the remainder of the command line.
50      * @param sh the current shell
51      *
52      * @exception JunoException for reporting errors
53      */
54
55     public abstract void doit( StringTokenizer args, Shell sh )
56

```

```

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

```

```

1 // fo1/7/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 7
19 */
20
21 public class ShellCommandTable
22 {
23     private Map table = new TreeMap();
24
25     /**
26      * Construct and fill a shell command table.
27      */
28
29     public ShellCommandTable()
30     {
31         fillTable();
32     }
33
34     /**
35      * Get a ShellCommand, given the command name key.
36      *
37      * @param key the name associated with the command we're
38      *         looking for.
39      *
40      * @return the command we're looking for, null if none.
41      */
42
43     public ShellCommand lookup( String key )
44     {
45         ShellCommand commandObject = (ShellCommand) table.get( key );
46         if (commandObject != null) {
47             return commandObject;
48         }
49
50         // try to load dynamically
51         // construct classname = "KeyCommand"
52         char[] chars = (key + "Command").toCharArray();
53         chars[0] = key.toUpperCase().charAt(0);
54         String classname = new String(chars);
55         try {
56             commandObject =

```

```

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

```



```

1 // fo1/7/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 7
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 // fo1/7/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 7
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                 + filename);
58         }
59     }
60
61     // EEE
62 }

```

```

57     }
58 }
59 }

```

```

// EEE

```

```
1 // fo1/7/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 7
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 // fo1/7/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 7
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 // fo1/7/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  *
15  * @version 7
16  */
17
18 public class ListCommand extends ShellCommand
19 {
20     // The constructor adds this object to the global table.
21
22     ListCommand()
23     {
24         super( "list contents of current directory" );
25     }
26
27     /**
28      * List contents of the current working directory.
29      *
30      * @param args the remainder of the command line.
31      * @param sh   the current shell
32      *
33      * @exception JunoException for reporting errors
34      */
35
36     public void doIt( StringTokenizer args, Shell sh )
37     throws JunoException
38     {
39         Terminal terminal = sh.getConsole();
40         Directory dir     = sh.getDot();
41         String[] fileNames = dir.getFileNames();
42
43         terminal.println( dir.getPathName() );
44         for ( int i = 0; i < fileNames.length; i++ ) {
45             String fileName = fileNames[i];
46             JFile jfile     = dir.retrieveJFile( fileName );
47             terminal.println( jfile.toString() );
48         }
49     }
50 }
```

```
1 // fo1/7/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 7
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     }
39 }
```

```
1 // fo1/7/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 7
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     {
36         throws JunoException
37     {
38         String filename = args.nextToken();
39         sh.getDot().removeFile(filename);
40     }
41 }
```



```

1 // fo1/7/files/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 7
23  */
24
25 public abstract class JFile
26 {
27     /**
28      * The separator used in pathnames.
29      */
30
31     public static final String separator = File.separator;
32
33     private String name; // a JFile knows its name
34     private User owner; // the owner of this file
35     private Date createDate; // when this file was created
36     private Date moddate; // when this file was last modified
37     private Directory parent; // the Directory containing this file
38
39     /**
40      * Construct a new JFile, set owner, parent, creation and
41      * modification dates. Add this to parent (unless this is the
42      * root Directory).
43      *
44      * @param name the name for this file (in its parent directory).
45      * @param creator the owner of this new file.
46      * @param parent the Directory in which this file lives.
47      */
48
49     protected JFile( String name, User creator, Directory parent )
50     {
51         this.name = name;
52         this.owner = creator;
53         this.parent = parent;
54         if (parent != null) {
55             parent.addJFile( name, this );
56

```

```

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

```

```

1 // fo1/7/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 7
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 // fo1/7/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 7
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 // fo1/7/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 7
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 }

```

```
1 // foj/7/juno/JunoException.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A general Juno Exception.
8  *
9  * @version 7
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/7/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 7
10 */
11
12 class BadShellCommandException extends JunoException
13 {
14     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 // fo1/7/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 7
10 */
11
12 public class ExitShellException extends JunoException
13 {
14 }
```