```java
// joi/10/juno/Juno.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

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

/**
 * Juno (Juno’s Unix NOt) mimics a command line operating system
 * such as Unix.
 * <p>
 * A Juno system has a name, a set of Users, a JFile system,
 * a login process and a set of shell commands.
 *
 * @see User
 * @see JFile
 * @see ShellCommand
 *
 * @version 10
 **/

public class Juno
    implements Serializable
{
    private final static String OS      = "Juno";
    private final static String VERSION = "10";

    private String    hostName;   // host machine name
    private Map       users;      // lookup table for Users
    private transient OutputInterface  console;

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

    private ShellCommandTable commandTable; // shell commands

    // file containing Juno state
    private transient String fileName = null;

    // port used by Juno server for remote login
    private int junoPort = 2001;

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

    public Juno( String hostName, boolean echoInput,
                 boolean isGUI, boolean isRemote )
    {
```

```java
        // Initialize the Juno environment ...

        this.hostName     = hostName;
        users             = new TreeMap();
        commandTable      = new ShellCommandTable();

        // the file system

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

    // Set up the correct console:
    // command line (default), graphical or remote.

    private void setupConsole( boolean echoInput, boolean isGUI,
                               boolean isRemote )
    {
        LoginInterpreter interpreter
            = new LoginInterpreter( this, null );

        if (isGUI) {
            console = new GUILoginConsole( hostName,
                                   this, interpreter, echoInput);
        }
        else if (isRemote) {
            console = new RemoteConsole( this, echoInput, junoPort );
        }
        else {
            console = new JunoTerminal( echoInput );
        }

        // Tell the interpreter about the console
        interpreter.setConsole( console );

        // If we’re using a simple command line interface,
        // start that.  (Constructing a GUI starts the GUI.)
        // Shut down Juno when done

        if (!isGUI && !isRemote) {
            interpreter.CLILogin();
            shutDown();
        }
    }

    /**
     * Shut down this Juno system.
     *
     * Save state if required.
     */

    public void shutDown( )
    {
        if (fileName != null) {
            writeJuno( );
        }
    }

    /**
     * Set the name of file in which system state is kept.
     *
     * @param fileName the file name.
     */

    public void setFileName(String fileName)
    {
        this.fileName = fileName;
    }

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

    public String getHostName()
    {
        return hostName;
    }

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

    public String getOS()
    {
        return OS;
    }

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

    public String getVersion()
    {
        return VERSION;
    }

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

```java

    public Directory getUserHomes()
    {
        return userHomes;
    }

    /**
     * The shell command table for this system.
     *
     * @return the shell command table.
     */

    public ShellCommandTable getCommandTable()
    {
        return commandTable;
    }

    /**
     * Look up a user by user name.
     *
     * @param username the user’s name.
     * @return the appropriate User object.
     */

    public User lookupUser( String username )
    {
        return (User) users.get( username );
    }

    /**
     * Create a new User.
     *
     * @param userName the User’s login name.
     * @param home her home Directory.
     * @param password her password.
     * @param realName her real name.
     * @return newly created User.
     */

    public User createUser( String userName, Directory home,
                            String password, String realName )
    {
        User newUser = new User( userName, password,
                                 home, realName );
        users.put( userName, newUser );
        return newUser;
    }

    /**
     * The Juno system may be given the following command line
     * arguments:
     *
     * -e:        Echo all input (useful for testing).
     *
     * -version:  Report the version number and exit.
     *
     * -g:         Support a GUI console.
     *
     * -remote     Start Juno server.
     *
     * -f filename File to read/write system state from/to
     *
     * [hostname]: The name of the host on which
     *             Juno is running (optional).
     */

    public static void main( String[] args )
    {
        // Parse command line options

        boolean echoInput    = false;
        boolean versionQuery = false;
        boolean isGUI        = false;
        boolean isRemote     = false;
        String  hostName     = "mars";
        String  junoFileName = null;

        for (int i=0; i < args.length; i++) {
            if (args[i].equals("-e")) {
                echoInput = true;
            }
            else if (args[i].equals("-version")) {
                versionQuery = true;
            }
            else if (args[i].equals("-g")) {
                isGUI = true;
            }
            else if (args[i].equals( "-remote" )) {
                isRemote = true;
            }
            else if (args[i].equals("-f")) {
                junoFileName = args[++i];
            }
            else {
                hostName = args[i];
            }
        }

        // If it’s  a version query give the version and exit
        if ( versionQuery ) {
            System.out.println( OS + " version " + VERSION );
            System.exit(0);
        }

        // Create a new Juno or read one from a file.
        Juno junoSystem = null;
        if (junoFileName != null) {
            junoSystem = readJuno( junoFileName );
        }
        if (junoSystem == null) {
            junoSystem = new Juno( hostName, echoInput,
                                   isGUI, isRemote );
```

```java
        }
        junoSystem.setFileName( junoFileName );
        junoSystem.setupConsole( echoInput, isGUI, isRemote );
    }

    // Read Juno state from a file.
    //
    // @param junoFileName the name of the file containing the system.
    // @return the system, null if file does not exist.

    private static Juno readJuno(String junoFileName)
    {
        File file = new File( junoFileName );
        if (!file.exists()) {
            return null;
        }
        ObjectInputStream inStream = null;
        try {
            inStream = new ObjectInputStream(
                               new FileInputStream( file ) );
            Juno juno = (Juno)inStream.readObject();
            System.out.println(
               "Juno state read from file " + junoFileName);
            return juno;
        }
        catch (Exception e ) {
            System.err.println("Problem reading " + junoFileName );
            System.err.println(e);
            System.exit(1);
        }
        finally {
            try {
                inStream.close();
            }
            catch (Exception e) {
            }
        }
        return null; // you can never get here
    }

    // Write Juno state to a file.

    private void writeJuno()
    {
        ObjectOutputStream outStream = null;
        try {
            outStream = new ObjectOutputStream(
                               new FileOutputStream( fileName ) );
            outStream.writeObject( this );
            System.out.println(
               "Juno state written to file " + fileName);
        }
        catch (Exception e ) {
            System.err.println("Problem writing " + fileName);
            System.err.println(e);
        }
        finally {
            try {
                outStream.close();
            }
            catch (Exception e ) {
            }
        }
    }
}
```

```java
// joi/10/juno/LoginInterpreter.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Interpreter for Juno login commands.
 *
 * There are so few commands that if-then-else logic is OK.
 *
 * @version 10
 */

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

    private Juno system;             // the Juno object
    private OutputInterface console; // where output goes

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

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

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

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

    /**
     * Simulates behavior at login: prompt.
     */

    public void CLILogin()
    {
```

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


    /**
     * Parse user’s command line and dispatch appropriate
     * semantic action.
     *
     * @param  inputLine the User’s instructions.
     * @return true except for "exit" command
     *         or null inputLine.
     */

    public boolean interpret( String inputLine )
    {
        if (inputLine == null) {
            return false;
        }
        StringTokenizer st =
            new StringTokenizer( inputLine );
        if (st.countTokens() == 0) {
            return true; // skip blank line
        }
        String visitor = st.nextToken();
        if (visitor.equals( "exit" )) {
            return false;
        }
        if (visitor.equals( "register" )) {
            register( st );
        }
        else if (visitor.equals( "help" )) {
            help();
        }
        else {
            String password;
            try {
                if (console.isGUI()) {
                    password = st.nextToken();
                }
                else {;
                    password = readPassword( "password: " );
                }
                User user = system.lookupUser(visitor);
                user.matchPassword( password );
                new Shell( system, user, console );
            }
            catch (Exception e) {

                // NullPointerException if no such user,
                // JunoException if password fails to match -
                // message to user doesn’t give away which.
                // The sysadmin would probably want a log
                // that did keep track.
                //
                // Other exceptions should be caught in shell()

                console.println("sorry");
            }
        }
        return true;
    }

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

    private void register( StringTokenizer line )
    {
        String username = "";
        String password = "";
        String realname = "";
        try {
            username = line.nextToken();
            password = line.nextToken();
            realname = line.nextToken("").trim();
        }
        catch (NoSuchElementException e) {
        }

        if (username.equals("") || password.equals("")
            || realname.equals("") ) {
            console.println(
                  "please supply username, password, real name");
            return;
        }
        User user = system.lookupUser(username);

        if (user != null) {  // user already exists
            console.println("sorry");
            return;
        }
        if (badPassword( password )) {
            console.println("password too easy to guess");
            return;
        }
        Directory home = new Directory( username, null,
                                        system.getUserHomes() );

        user = system.createUser( username, home, password, realname );
        home.setOwner( user );
    }

    // test to see if password is unacceptable:
    //   fewer than 6 characters
    //   contains only alphabetic characters
```

```java

    private boolean badPassword( String pwd )
    {
        if (pwd.length() < 6) {
            return true;
        }
        int nonAlphaCount = 0;
        for (int i=0; i < pwd.length(); i++) {
            if (!Character.isLetter(pwd.charAt(i))) {
                nonAlphaCount++;
            }
        }
        return (nonAlphaCount == 0);
    }

    // Used for reading the user’s password in CLI.

    private String readPassword( String prompt )
    {
        String line =
            ((InputInterface) console).readLine( prompt );
        StringTokenizer st = new StringTokenizer( line );
        try {
            return st.nextToken();
        }
        catch ( NoSuchElementException e )  {}
        return ""; // keeps compiler happy
    }


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

    private void welcome()
    {
        console.println( "Welcome to " + system.getHostName() +
                         " running "   + system.getOS() +
                         " version "   + system.getVersion() );
        help();
    }

    // Remind user of available commands.

    private void help()
    {
        console.println( LOGIN_COMMANDS );
        console.println("");
    }
}
```

```java
// joi/10/juno/Shell.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * Models a shell (command interpreter)
 *
 * The Shell knows the (Juno) system it’s working in,
 * the User who started it,
 * and the console to which to send output.
 *
 * It keeps track of the the current working directory (.) .
 *
 * @version 10
 */

public class Shell
    implements InterpreterInterface
{
    private Juno system;             // The operating system object
    private User user;               // The user logged in
    private OutputInterface console; // The console for this shell
    private Directory dot;           // The current working directory

    /**
     * Construct a login shell for the given user and console.
     *
     * @param system a reference to the Juno system.
     * @param user the User logging in.
     * @param console a Terminal for input and output.
     */

    Shell( Juno system, User user, OutputInterface console )
    {
        this.system  = system;
        this.user    = user;
        this.console = console;
        dot          = user.getHome(); // default current directory

        if (!console.isGUI()) {
            this.console = console;
            CLIShell();
        }
        else
            this.console =
                new GUIShellConsole("Juno shell for " + user,
                                    this, console.isEchoInput());
    }

    // Run the command line interpreter

    private void CLIShell()
    {
```

```java
        boolean moreWork = true;
        while(moreWork) {
            moreWork = interpret( ((InputInterface) console).
                                  readLine( getPrompt() ) );
        }
        console.println("goodbye");
    }

    /**
     * Interpret a String.
     *
     * Syntax
     * <pre>
     *    shellcommand command-arguments
     * </pre>
     *
     * @param inputLine the String to interpret.
     * @return true unless shell command is logout.
     */

    public boolean interpret( String inputLine )
    {
        StringTokenizer st = stripComments(inputLine);
        if (st.countTokens() == 0) {           // skip blank line
            return true;
        }
        String commandName = st.nextToken();
        ShellCommand commandObject =
            system.getCommandTable().lookup( commandName );
        if (commandObject == null ) {
            console.errPrintln( "Unknown command: " + commandName );
            return true;
        }
        try {
            commandObject.doIt( st, this );
        }
        catch (ExitShellException e) {
            return false;
        }
        catch (BadShellCommandException e) {
            console.errPrintln( "Usage: " + commandName + " " +
                                e.getCommand().getArgString() );
        }
        catch (JunoException e) {
            console.errPrintln( e.getMessage() );
        }
        catch (Exception e) {
            console.errPrintln( "you should never get here" );
            console.errPrintln( e.toString() );
        }
        return true;
    }

    // Strip characters from ’#’ to end of line, create and
    // return a StringTokenizer for what’s left.

    private StringTokenizer stripComments( String line )
    {
        int commentIndex = line.indexOf('#');
        if (commentIndex >= 0) {
            line = line.substring(0,commentIndex);
        }
        return new StringTokenizer(line);
    }

    /**
     * The prompt for the CLI.
     *
     * @return the prompt string.
     */

    public String getPrompt()
    {
      return system.getHostName() + ":" + getDot().getPathName() + "> ";
    }

    /**
     * The User associated with this shell.
     *
     * @return the user.
     */

    public User getUser()
    {
        return user;
    }

    /**
     * The current working directory for this shell.
     *
     * @return the current working directory.
     */

    public Directory getDot()
    {
        return dot;
    }

    /**
     * Set the current working directory for this Shell.
     *
     * @param dot the new working directory.
     */

    public void setDot(Directory dot)
    {
        this.dot = dot;
    }


    /**
     * The console associated with this Shell.
```

```
     *
     * @return the console.
     */

    public OutputInterface getConsole()
    {
        return console;
    }

    /**
     * The Juno object associated with this Shell.
     *
     * @return the Juno instance that created this Shell.
     */

    public Juno getSystem()
    {
        return system;
    }
}
```

```java
// joi/10/juno/ShellCommand.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * Model those features common to all ShellCommands.
 *
 * Each concrete extension of this class provides a constructor
 * and an implementation for method doIt.
 *
 * @version 10
 */

public abstract class ShellCommand
    implements java.io.Serializable
{
    private String helpString;   // documents the command
    private String argString;    // any args to the command

    /**
     * A constructor, always called (as super()) by the subclass.
     * Used only for commands that have arguments.
     *
     * @param helpString a brief description of what the command does.
     * @param argString a prototype illustrating the required arguments.
     */

    protected ShellCommand( String helpString, String argString )
    {
        this.argString  = argString;
        this.helpString = helpString;
    }

    /**
     * A constructor for commands having no arguments.
     *
     * @param helpString a brief description of what the command does.
     */

    protected ShellCommand( String helpString )
    {
        this( helpString, "" );
    }

    /**
     * Execute the command.
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     *
     * @exception JunoException for reporting errors
     */

```

```java
    public abstract void doIt( StringTokenizer args, Shell sh )
        throws JunoException;

    /**
     * Help for this command.
     *
     * @return the help string.
     */

    public String getHelpString()
    {
        return helpString;
    }

    /**
     * The argument string prototype.
     *
     * @return the argument string prototype.
     */

    public String getArgString()
    {
        return argString;
    }
}
```

```java
// joi/10/juno/ShellCommandTable.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * A ShellCommandTable object maintains a dispatch table of
 * ShellCommand objects keyed by the command names used to invoke
 * them.
 *
 * To add a new shell command to the table, install it from
 * method fillTable().
 *
 * @see ShellCommand
 *
 * @version 10
 */

public class ShellCommandTable
    implements java.io.Serializable
{
    private Map table = new TreeMap();

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

    public ShellCommandTable()
    {
        fillTable();
    }

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

    public ShellCommand lookup( String key )
    {
        ShellCommand commandObject = (ShellCommand) table.get( key );
        if (commandObject != null) {
            return commandObject;
        }

        // try to load dynamically
        // construct classname = "KeyCommand"
        char[] chars = (key + "Command").toCharArray();
        chars[0] = key.toUpperCase().charAt(0);
        String classname = new String(chars);
        try {
```

```java
            commandObject =
                (ShellCommand)Class.forName(classname).newInstance();
        }
        catch (Exception e) { // couldn’t find class
            return null;
        }
        install(key, commandObject); // put it in table for next time
        return commandObject;
    }

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

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

    // Associate a command name with a ShellCommand.

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

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

    private void fillTable()
    {
        install( "list", new ListCommand() );
        install( "cd", new CdCommand() );
        install( "newfile", new NewfileCommand() );
        install( "remove", new RemoveCommand() );
        install( "help", new HelpCommand() );
        install( "mkdir", new MkdirCommand() );
        install( "type", new TypeCommand() );
        install( "logout", new LogoutCommand() );
    }
}
```

```java
// joi/10/juno/MkdirCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to create a new directory.
 * Usage:
 * <pre>
 *     mkdir directory-name
 * </pre>
 *
 * @version 10
 */

public class MkdirCommand extends ShellCommand
{
    MkdirCommand()
    {
        super( "create a subdirectory of the current directory",
               "directory-name" );
    }

    /**
     * Create a new Directory in the current Directory.
     *
     * @param args the remainder of the command line.
     * @param sh the current shell.
     *
     * @exception JunoException for reporting errors.
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        String filename = args.nextToken();
        new Directory( filename, sh.getUser(), sh.getDot() );
    }
}
```

```java
// joi/10/juno/TypeCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to display the contents of a
 * text file.
 * Usage:
 * <pre>
 *     type textfile
 * </pre>
 *
 * @version 10
 */

public class TypeCommand extends ShellCommand
{
    TypeCommand()
    {
        super( "display contents of a TextFile", "textfile" );
    }

    /**
     * Display the contents of a TextFile.
     *
     * @param args the remainder of the command line.
     * @param sh the current Shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
        throws JunoException
    {
        String filename;
        try {
            filename = args.nextToken();
        }
        catch (NoSuchElementException e) {
            throw new BadShellCommandException( this );
        }
        try {
            sh.getConsole().println(
                ( (TextFile) sh.getDot().
                  retrieveJFile( filename ) ).getContents() );
        }
        catch (NullPointerException e) {
            throw new JunoException( "JFile does not exist: "
                                     + filename);
        }
        catch (ClassCastException e) {
            throw new JunoException( "JFile not a text file: "
                                     + filename);
```

```java
        }
    }
}
```

```java
// joi/10/juno/HelpCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to display help on the shell commands.
 * Usage:
 * <pre>
 *     help
 * </pre>
 *
 * @version 10
 */

public class HelpCommand extends ShellCommand
{
    HelpCommand()
    {
        super( "display ShellCommands" );
    }

    /**
     * Print out help for all commands.
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        // Get command keys from global table, print them out.

        sh.getConsole().println( "shell commands" );
        ShellCommandTable table = sh.getSystem().getCommandTable();
        String[] names = table.getCommandNames();
        for (int i = 0; i < names.length; i++ ) {
            String cmdname = names[i];
            ShellCommand cmd =
                (ShellCommand) table.lookup( cmdname );
            sh.getConsole().
                println( "  " + cmdname + ": " + cmd.getHelpString() );
        }
    }
}
```

```java
// joi/10/juno/NewfileCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to create a text file.
 * Usage:
 * <pre>
 *     newfile filename contents
 * </pre>
 *
 * @version 10
 */

public class NewfileCommand extends ShellCommand
{
    NewfileCommand()
    {
        super( "create a new TextFile", "filename contents" );
    }

    /**
     * Create a new TextFile in the current Directory.
     *
     * @param args the remainder of the command line.
     * @param sh the current shell.
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        String filename;
        String contents;
        filename = args.nextToken();
        contents  = args.nextToken("").trim(); // rest of line
        new TextFile( filename, sh.getUser(),
                      sh.getDot(), contents );
    }
}
```

```java
// joi/10/juno/CdCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to change directory.
 * Usage:
 * <pre>
 *     cd [directory]
 * </pre>
 * for moving to the named directory.
 *
 * @version 10
 */

class CdCommand extends ShellCommand
{
    CdCommand()
    {
        super( "change current directory", "[ directory ]" );
    }

    /**
     * Move to the named directory
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        String dirname = "";
        Directory d = sh.getUser().getHome(); // default
        if ( args.hasMoreTokens() ) {
            dirname = args.nextToken();
            if (dirname.equals("..")) {
                if (sh.getDot().isRoot())
                    d = sh.getDot(); // no change
                else
                    d = sh.getDot().getParent();
            }
            else if (dirname.equals(".")) {
                d = sh.getDot(); // no change
            }
            else {
                d = (Directory)(sh.getDot().retrieveJFile(dirname));
            }
        }
        sh.setDot( d );
    }
```

```java
}
```

```java
// joi/10/juno/ListCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to list contents of the current directory.
 * Usage:
 * <pre>
 *     list
 * </pre>
 *
 * @version 10
 */

public class ListCommand extends ShellCommand
{
    // The constructor adds this object to the global table.

    ListCommand()
    {
        super( "list contents of current directory" );
    }

    /**
     * List contents of the current working directory.
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
        throws JunoException
    {
        OutputInterface terminal = sh.getConsole();
        Directory dir            = sh.getDot();
        String[] fileNames       = dir.getFileNames();

        terminal.println( dir.getPathName() );
        for ( int i = 0; i < fileNames.length; i++ ) {
            String fileName = fileNames[i];
            JFile  jfile    = dir.retrieveJFile( fileName );
            terminal.println( jfile.toString() );
        }
    }
}
```

```java
// joi/10/juno/GetfileCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

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

/**
 * The Juno shell command to get a text file from the underlying
 * operating system and copy it to a Juno text file.
 * Usage:
 * <pre>
 *     getfile native-filename juno-filename
 * </pre>

 * <pre>
 *
 * @version 10
 */

class GetfileCommand extends ShellCommand
{
    GetfileCommand()
    {
        super( "download a file to Juno",
               "native-filename juno-filename" );
    }

    /**
     * Use the getfile command to copy the content of a real
     * file to a Juno TextFile.
     * <p>
     * The command has the form:
     * <pre>
     * get nativeFile textfile <&>
     *
     * @param args: the reminder of the command line.
     * @param sh: the current shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        if ( sh.getConsole().isRemote() ) {
            throw( new JunoException(
                "Get not implemented for remote consoles." ) );
        }
        String src;
        String dst;
        try {
            src = args.nextToken();
            dst = args.nextToken();
        }
```

```java
        catch (NoSuchElementException e) {
            throw new BadShellCommandException( this );
        }
        BufferedReader inStream  = null;
        Writer         outStream = null;
        try {
            inStream  = new BufferedReader( new FileReader( src ) );
            outStream = new StringWriter();
            String line;

            while ((line = inStream.readLine()) != null) {
                outStream.write( line );
                outStream.write( '\n' );
            }
            new TextFile( dst, sh.getUser(),
                          sh.getDot(), outStream.toString() );
        }
        catch (IOException e) {
            throw new JunoException( "IO problem in get" );
        }
        finally {
            try {
                inStream.close();
                outStream.close();
            }
            catch (IOException e) {};
        }
    }
}
```

```java
// joi/10/juno/RemoveCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to remove a text file.
 * Usage:
 * <pre>
 *     remove textfile
 * </pre>
 *
 * @version 10
 */

public class RemoveCommand extends ShellCommand
{
    RemoveCommand()
    {
        super( "remove a TextFile", "textfile" );
    }

    /**
     * Remove a TextFile.
     *
     * @param args the remainder of the command line.
     * @param sh   the current Shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        String filename = args.nextToken();
        sh.getDot().removeJFile(filename);
    }
}
```

```java
// joi/10/juno/LogoutCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to log out.
 * Usage:
 * <pre>
 *     logout
 * </pre>
 *
 * @version 10
 */

public class LogoutCommand extends ShellCommand
{
    LogoutCommand()
    {
        super( "log out, return to login: prompt" );
    }

    /**
     * Log out from the current shell.
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        throw new ExitShellException();
    }
}
```

```java
// joi/10/jfiles/JFile.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

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

/**
 * A JFile object models a file in a hierarchical file system.
 * <p>
 * Extend this abstract class to create particular kinds of JFiles,
 * e.g.:<br>
 * Directory  -
 *    a JFile that maintains a list of the files it contains.<br>
 * TextFile  -
 *    a JFile containing text you might want to read.<br>
 *
 * @see Directory
 * @see TextFile
 *
 * @version 10
 */

public abstract class JFile
    implements java.io.Serializable
{
    /**
     * The separator used in pathnames.
     */

    public static final String separator = File.separator;

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

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

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

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

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

    public String getName()
    {
        return name;
    }

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

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

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

    public abstract int getSize();

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

    public abstract String getSuffix();

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

    public void setOwner( User owner )
    {
        this.owner = owner;
    }

    /**
     * The file’s owner.
     *
     * @return the owner of the file.
     */

    public User getOwner()
    {
        return owner;
    }

    /**
     * The date and time of the file’s creation.
     *
     * @return the file’s creation date and time.
     */

    public String getCreateDate()
    {
        return createDate.toString();
    }

    /**
     * Set the modification date to "now".
     */

    protected void setModDate()
    {
        modDate = new Date();
    }

    /**
     * The date and time of the file’s last modification.
     *
     * @return the date and time of the file’s last modification.
     */

    public String getModDate()
    {
        return modDate.toString();
    }

    /**
     * The Directory containing this file.
     *
     * @return the parent directory.
     */

    public Directory getParent()
    {
        return parent;
```

```java
    }

    /**
     * A JFile whose parent is null is defined to be the root
     * (of a tree).
     *
     * @return true when this JFile is the root.
     */

    public boolean isRoot()
    {
        return (parent == null);
    }

    /**
     * How a JFile represents itself as a String.
     * That is,
     * <pre>
     *    owner   size    modDate    name+suffix
     * </pre>
     *
     * @return the String representation.
     */

    public String toString()
    {
        return getOwner() + "\t" +
            getSize()  + "\t" +
            getModDate()  + "\t" +
            getName() + getSuffix();
    }
}
```

```java
// joi/10/juno/Directory.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Directory of JFiles.
 *
 * A Directory is a JFile that maintains a
 * table of the JFiles it contains.
 *
 * @version 10
 */

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

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

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

    /**
     * The size of a Directory is the number of JFiles it contains.
     *
     * @return the Directory’s size.
     */

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

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

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

```java
    }

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

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

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

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

    /**
     * Remove a JFile in this Directory, by name .
     *
     * @param filename the name of the JFile to remove
     */

    public void removeJFile( String filename )
    {
        jfiles.remove( filename );
    }

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

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

```java
// joi/10/juno/TextFile.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * A TextFile is a JFile that holds text.
 *
 * @version 10
 */

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

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

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

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

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

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

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

    /**
```

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

    public String getSuffix()
    {
        return "";
    }

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

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

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

    public String getContents()
    {
        return contents;
    }

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

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


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

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

```
// joi/10/juno/User.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * Model a juno user.  Each User has a login name, password,
 * a home directory, and a real name.
 * name.
 *
 * @version 10
 */

public class User
    implements java.io.Serializable
{
    private String name;     // the User’s login name
    private String password; // The user’s login password.
    private Directory home;  // her home Directory
    private String realName; // her real name

    /**
     * Construct a new User.
     *
     * @param name      the User’s login name.
     * @param password  the user’s login password.
     * @param home      her home Directory.
     * @param realName  her real name.
     */

    public User( String name, String password,
                 Directory home, String realName )
    {
        this.name     = name;
        this.password = password;
        this.home     = home;
        this.realName = realName;
    }

    /**
     *   Confirm password. Throw a JunoException on failure.
     *
     *   @param guess the string to test against the password.
     *
     *   @exception JunoException
     *      if password fails to match
     */

    public void matchPassword( String guess ) throws JunoException
    {
        if (!guess.equals( password )) {
            throw new JunoException( "bad password" );
        }
    }

    /**
```

```
     * Get the User’s login name.
     *
     * @return the name.
     */

    public String getName()
    {
        return name;
    }

    /**
     * Convert the User to a String.
     * The String representation for a User is her
     * login name.
     *
     * @return the User’s name.
     */

    public String toString()
    {
        return getName();
    }

    /**
     * Get the User’s home Directory.
     *
     * @return the home Directory.
     */

    public Directory getHome()
    {
        return home;
    }

    /**
     * Get the user’s real name.
     *
     * @return the real name.
     */

    public String getRealName()
    {
        return realName;
    }
}
```

```java
// joi/10/juno/JunoException.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A general Juno Exception.
 *
 * @version 10
 */

public class JunoException extends Exception
{
    /**
     * The default (no argument) constructor.
     */

    public JunoException()
    {
    }

    /**
     * A general Juno exception holding a String message.
     *
     * @param message the message.
     */

    public JunoException( String message )
    {
        // Exception (actually Throwable, Exceptions’s superclass)
        // can remember the String passed its constructor.

        super( message );
    }

    // Note, to get the message stored in a JunoException
    // we can just use the (inherited) methods getMessage(),
    // and toString().
}
```

```java
// joi/10/juno/BadShellCommandException.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * The Exception generated when a ShellCommand is misused.
 *
 * @version 10
 */

class BadShellCommandException extends JunoException
{
    private ShellCommand command;

    /**
     * Construct a new BadShellCommandException
     * containing the badly used command.
     *
     * @param the ShellCommand being misused.
     */

    public BadShellCommandException( ShellCommand command )
    {
        this.command = command;
    }

    /**
     * Get the command.
     */

    public ShellCommand getCommand()
    {
        return command;
    }
}
```

```java
// joi/10/juno/ExitShellException.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * Exception raised for exiting a shell.
 *
 * @version 10
 */

public class ExitShellException extends JunoException
{
}
```

```java
// joi/10/juno/ShellCommandNotFoundException.java (version 10)
//
//
// Copyright 1997-2001 Ethan Bolker and Bill Campbell

/**
 * The Exception when a ShellCommand isn't found.
 */

class ShellCommandNotFoundException extends JunoException
{
    /**
     * Create a ShellCommandNotFoundException.
     */

    public ShellCommandNotFoundException()
    {
    }

    /**
     * Create a ShellCommandNotFoundException with
     * a message reporting the command tried.
     */

    public ShellCommandNotFoundException(String commandName )
    {
        super( "ShellCommand " + commandName + " not found" );
    }
}
```

```java
// joi/10/juno/JFileNotFoundException.java (version 10)
//
//
// Copyright 1997-2001 Ethan Bolker and Bill Campbell

/**
 * The Exception thrown when a JFile isn’t found
 *
 * @version 10
 */

class JFileNotFoundException extends JunoException
{
    String jfilename;

    /**
     * Construct a new JFileNotFoundException
     *
     * @param jfilename the file sought
     */

    public JFileNotFoundException( String jfilename )
    {
        super( "JFile " + jfilename + " not found." );
        this.jfilename = jfilename;
    }

    /**
     * Get the name of the file that wasn’t there.
     *
     * @return the file name
     */

    public String getJFilename()
    {
        return jfilename;
    }
}
```

```java
// joi/10/juno/GUILoginConsole.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

/**
 *  The graphical user interface to Juno.
 */

public class GUILoginConsole extends JFrame
    implements OutputInterface
{
    private static final int FIELDWIDTH = 30;
    private static final int FIELDHEIGHT = 5;

    private final Juno junoSystem;
    private WindowCloser closeMe; // to shut down Juno

    private String title;  // title for the windows

    // The interpreter interprets one-line commands.
    private InterpreterInterface interpreter;
    private boolean echoInput;

    // All output goes to messages.
    private JTextArea messages;

    /**
     *  Construct a GUI console for Juno.
     *
     *  @param title the title for this window.
     *  @param junoSystem the Juno system for which this is a GUI
     *  @param interpreter the object to which to send user input.
     *  @param echoInput true when input echoes to this console.
     */

    public GUILoginConsole( String title, Juno junoSystem,
                            InterpreterInterface interpreter,
                            boolean echoInput)
    {
        super( title );
        this.title       = title;
        this.junoSystem  = junoSystem;
        this.interpreter = interpreter;
        this.echoInput   = echoInput;
        this.closeMe = new WindowCloser( junoSystem );

        // Set up the look and feel;
        // Everthing is placed on a panel (using BorderLayout)

        JPanel panel = new JPanel();
```

```java
        panel.setLayout( new BorderLayout() );

        // First a tabbed pane, with two tabs:
        //   one for login, one for registration

        JTabbedPane tabs = new JTabbedPane();
        tabs.addTab( "Login" , null,
                     new LoginPane( interpreter, echoInput, closeMe ) );
        tabs.addTab( "Register", null,
                     new RegisterPane( interpreter, echoInput ) );
        tabs.setSelectedIndex( 0 );  // Login selected by default
        panel.add( tabs, BorderLayout.NORTH );

        // and the output messages area.
        panel.add( new JLabel( "Messages:" ), BorderLayout.CENTER );
        messages = new JTextArea( FIELDHEIGHT, FIELDWIDTH );
        panel.add( messages, BorderLayout.SOUTH );

        // Add the panel to this JFrame
        this.getContentPane().add( panel );

        // Closing this window
        this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        this.addWindowListener( closeMe );

        // Size and display this JFrame
        pack();
        show();
    }

    // Implementing the OutputInterface. Everything goes to the
    // single message area.

    /**
     * Write a String followed by a newline
     * to message area.
     *
     * @param str - the string to write
     */

    public void println(String str )
    {
        messages.append( str + "\n" );
    }

    /**
     * Write a String followed by a newline
     * to message area.
     *
     * @param str - the String to write
     */

    public void errPrintln(String str )
    {
        println( str );
    }

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if echoing input.
     */

    public boolean isEchoInput()
    {
        return echoInput;
    }

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if GUI
     */

    public boolean isGUI()
    {
        return true;
    }

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if remote
     */

    public boolean isRemote()
    {
        return false;
    }

    // The Login pane is specified in a private inner class,
    // visible only here.

    private class  LoginPane extends JPanel
    {
        // The login pane has two text fields and two buttons.
        private JTextField nameField;
        private JTextField passwordField;

        private JButton ok;
        private JButton exit;

        private WindowCloser closeMe; // to shut down Juno
        // Construct the login pane and set up its listeners.

        public LoginPane( InterpreterInterface interpreter,
                          boolean echoInput, WindowCloser closeMe )
        {
            super();
            this.closeMe = closeMe;

            // Set up the look and feel.
```

```java
            // Everything will go into a vertical Box, a container
            // whose contents are laid out using BoxLayout

            Box box = Box.createVerticalBox();

            // First a panel, containing the two text fields

            JPanel p  = new JPanel();
            p.setLayout( new GridLayout( 4 , 1 ) );

            p.add( new JLabel( "Login:" ) );
            nameField = new JTextField( FIELDWIDTH );
            p.add( nameField );

            p.add( new JLabel( "Password:" ) );
            passwordField = new JPasswordField( FIELDWIDTH );
            p.add( passwordField );
            box.add( p );
            box.add( Box.createVerticalStrut( 15 ) );

            // Then a horizontal Box containing the two buttons

            Box row = Box.createHorizontalBox();
            row.add( Box.createGlue() );

            ok = new JButton( "OK" );
            row.add( ok );
            row.add( Box.createGlue() );

            exit = new JButton( "Exit" );
            row.add( exit );
            row.add( Box.createGlue() );
            box.add( row );
            box.add( Box.createVerticalStrut( 15 ) );

            this.setLayout( new BorderLayout() );
            this.add( box, BorderLayout.CENTER );

            // Set up the listeners (the semantics)

            ok.addActionListener( new LoginProcessor() );
            exit.addActionListener( closeMe ); // shuts down Juno
        }

        // An inner inner class for the semantics
        // when the user clicks OK.

        private class LoginProcessor implements ActionListener
        {
            public void actionPerformed(ActionEvent e)
            {
                String str = nameField.getText() +  " " +
                    passwordField.getText();
                nameField.setText("");
                passwordField.setText("");
                messages.setText(str+'\n');  // for debugging
                interpreter.interpret( str );
            }
        }
    }

    // The Register pane is specified in a private inner class,
    // visible only here.

    private class  RegisterPane extends JPanel
    {
        // The register pane has four textfields and two buttons.
        private JTextField chosenName;
        private JTextField fullName;
        private JTextField password1;
        private JTextField password2;

        private JButton register;
        private JButton clear;

        public RegisterPane( InterpreterInterface interpreter,
                             boolean echoInput)
        {
            super();

            // Define the look and feel
            // Everything goes into a vertical Box
            Box box = Box.createVerticalBox();

            // First a panel containing the text fields

            JPanel p  = new JPanel();
            p.setLayout( new GridLayout( 0 , 1 ) );

            p.add( new JLabel( "Choose login name:" ) );
            chosenName = new JTextField( FIELDWIDTH );
            p.add( chosenName );

            p.add( new JLabel( "Give full name:" ) );
            fullName = new JTextField( FIELDWIDTH );
            p.add( fullName );

            p.add( new JLabel( "Choose password:" ) );
            password1  = new JPasswordField( FIELDWIDTH );
            p.add( password1 );

            p.add( new JLabel( "Retype password:" ) );
            password2  = new JPasswordField( FIELDWIDTH );
            p.add( password2 );

            box.add( p );
            box.add( Box.createVerticalStrut( 15 ) );

            // Then a horizontal Box containing the buttons

            Box row = Box.createHorizontalBox();
            row.add( Box.createGlue() );
```

```java
            register = new JButton( "Register" );
            row.add( register );
            row.add( Box.createGlue() );
            clear = new JButton( "Clear" );
            row.add( clear );
            row.add( Box.createGlue() );
            box.add( row );
            box.add( Box.createVerticalStrut( 15 ) );

            this.setLayout( new BorderLayout() );
            this.add( box, BorderLayout.CENTER );

            // Set up the listeners (the semantics)

            register.addActionListener( new Registration() );
            clear.addActionListener( new Cleanser() );
        }

        // An inner inner class for the semantics when the user
        // clicks Register.

        private class Registration implements ActionListener
        {
            public void actionPerformed(ActionEvent e)
            {
                if ( password1.getText().trim().equals(
                     password2.getText().trim() ) ) {
                    String str = "register " +
                        chosenName.getText() +  " " +
                        password1.getText() + " " +
                        fullName.getText() ;
                    chosenName.setText("");
                    fullName.setText("");
                    messages.setText(str+'\n');  // for debugging
                    interpreter.interpret(str);
                }
                else {
                    messages.setText(
                        "Sorry, passwords don't match.\n" );
                }
                password1.setText("");
                password2.setText("");
            }
        }

        // An inner inner class for the semantics when the user
        // clicks Clear.

        private class Cleanser implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                chosenName.setText("");
                fullName.setText("");
                password1.setText("");
                password2.setText("");
            }
        }
```

```
    }

    //  A WindowCloser instance handles close events generated
    //  by the underlying window system with its windowClosing
    //  method, and close events from buttons or other user
    //  components with its actionPerformed method.
    //
    //  The action is to shut down Juno.

    private static class WindowCloser extends WindowAdapter
        implements ActionListener
    {
        Juno system;

        public WindowCloser( Juno system )
        {
            this.system = system;
        }

        public void windowClosing (WindowEvent e)
        {
            this.actionPerformed( null );
        }

        public void actionPerformed(ActionEvent e)
        {
            if (system != null) {
                system.shutDown();
            }
            System.exit(0);
        }
    }

    /**
     * main() in GUILoginConsole class for
     * unit testing during development.
     */

    public static void main( String[] args )
    {
        new GUILoginConsole( "GUItest", null, null, true ).show();
    }
}
```

```java
// joi/10/juno/GUIShellConsole.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
 *  The GUI to the Juno system Shell.
 */

public class GUIShellConsole extends JFrame
    implements OutputInterface
{
    private static final int FIELDWIDTH  = 50;
    private static final int FIELDHEIGHT = 10;

    // the components on the window

    private JLabel promptLabel = new JLabel();
    private JTextField commandLine = new JTextField( FIELDWIDTH );
    private JButton doIt       = new JButton( "Do It" );
    private JButton logout     = new JButton( "Logout" );
    private JTextArea stdout   =
                         new JTextArea( FIELDHEIGHT, FIELDWIDTH );
    private JTextArea stderr   =
                         new JTextArea( FIELDHEIGHT/2, FIELDWIDTH );

    private Shell sh;    // for interpreting shell commands
    private WindowCloser closeMe; // for logging out.

    private boolean echoInput;

    /**
     * Construct a GUI console for a shell.
     *
     * @param title the title to  display in the frame.
     * @param sh the shell to interpret commands.
     * @param echoInput is input to be echoed?
     */

    public GUIShellConsole( String title,
                            Shell sh,
                            boolean echoInput )
    {
        this.sh = sh;
        this.echoInput   = echoInput;

        setTitle( title );
        setPrompt( sh.getPrompt() );

        // set up console's look and feel

```

```java
        JPanel outerPanel = new JPanel();
        outerPanel.setLayout( new BorderLayout() );

        Box box = Box.createVerticalBox();

        JPanel commandPanel = new JPanel();
        commandPanel.setLayout( new BorderLayout() );
        commandPanel.add( promptLabel, BorderLayout.NORTH );
        commandPanel.add( commandLine, BorderLayout.CENTER );
        box.add( commandPanel );
        box.add( Box.createVerticalStrut( 10 ) );

        Box buttons = Box.createHorizontalBox();
        buttons.add( Box.createGlue() );
        buttons.add( doIt );
        buttons.add( Box.createGlue() );
        buttons.add( logout );
        buttons.add( Box.createGlue() );
        box.add( buttons );
        box.add( Box.createVerticalStrut( 10 ) );

        JPanel stdoutPanel = new JPanel();
        stdoutPanel.setLayout( new BorderLayout() );
        stdoutPanel.add( new JLabel( "Standard output:"),
                         BorderLayout.NORTH );

        stdoutPanel.add( new JScrollPane( stdout ),
                         BorderLayout.CENTER );

        box.add( stdoutPanel );
        box.add( Box.createVerticalStrut( 10 ) );
        stdout.setEditable( false );

        JPanel stderrPanel = new JPanel();
        stderrPanel.setLayout( new BorderLayout() );
        stderrPanel.add( new JLabel( "Error output:" ),
                         BorderLayout.NORTH );
        stderrPanel.add( new JScrollPane( stderr ),
                         BorderLayout.CENTER );
        box.add( stderrPanel );
        box.add( Box.createVerticalStrut( 10 ) );
        stderr.setEditable( false );

        outerPanel.add( box, BorderLayout.CENTER );
        this.getContentPane().add( outerPanel, BorderLayout.CENTER );

        // Install menus and tool bar.

        JMenuBar menuBar  = new JMenuBar();
        JMenu commandMenu = new JMenu( "Command" );
        JMenu helpMenu    = new JMenu( "Help" );

        JToolBar toolBar  = new JToolBar();

        // Create menu items and tool buttons for each shell command

        ShellCommandTable table = sh.getSystem().getCommandTable();
        String [] commandNames = table.getCommandNames();
        for ( int i = 0; i < commandNames.length; i++ ) {

            String commandName = commandNames[i];
            ShellCommand command =
                table.lookup( commandName );

            CommandMenuAction commandAction =
                new CommandMenuAction( commandName,
                                       command.getArgString() );

            HelpMenuAction helpAction =
                new HelpMenuAction( commandName,
                                    command.getArgString(),
                                    command.getHelpString() );

            JMenuItem item1 = commandMenu.add( commandAction );
            JMenuItem item2 = helpMenu.add( helpAction );
            JButton button  = toolBar.add( commandAction );
            button.setToolTipText( command.getHelpString() );
        }

        this.setJMenuBar( menuBar );
        this.getContentPane().add( toolBar,
                                   BorderLayout.NORTH );
        menuBar.add( commandMenu );
        menuBar.add( helpMenu );

        pack();
        show();

        // add Listener to the Do It button

        doIt.addActionListener( new Interpreter() );

        // add listener to the Logout button and window closer

        closeMe = new WindowCloser( this );
        logout.addActionListener( closeMe );
        this.addWindowListener( closeMe );
    }

    // Set the GUI prompt

    private void setPrompt(String prompt)
    {
        this.promptLabel.setText(prompt);
    }

    // Implementing the OutputInterface.
    // Everything goes to the single message area.

    public void println( String str )
    {
        stdout.append(str + "\n");
```

```java
    }

    public void errPrintln( String str )
    {
        stderr.append(str + "\n");
    }

    public boolean isGUI()
    {
        return true;
    }

    public boolean isRemote()
    {
        return false;
    }

    public boolean isEchoInput()
    {
        return echoInput;
    }

    // An inner class for the semantics when the user submits
    // a ShellCommand for execution.

    private class Interpreter
        implements ActionListener
    {
        public void actionPerformed( ActionEvent e )
        {
            String str = commandLine.getText();
            stdout.append( sh.getPrompt() + str + '\n');
            if (sh.interpret( str )) {
                setPrompt( sh.getPrompt() );
            }
            else {
                closeMe.actionPerformed(null);
            }
        }
    }

    private class CommandMenuAction extends AbstractAction
    {
        private String argString;
        private String helpString;

        public CommandMenuAction( String text, String argString )
        {
            super( text );
            this. argString = argString;
        }

        public void actionPerformed( ActionEvent e )
        {
            commandLine.setText( getValue( Action.NAME ) +
                                 " " + argString );
        }
    }

    private class HelpMenuAction extends AbstractAction
    {
        private String argString;
        private String helpString;

        public HelpMenuAction( String text, String argString,
                               String helpString )
        {
            super( text );
            this. argString = argString;
            this.helpString = helpString;
        }

        public void actionPerformed( ActionEvent e )
        {
            stdout.append( getValue( Action.NAME ) + ": " +
                           helpString );
        }
    }

    //  A WindowCloser instance handles close events generated
    //  by the underlying window system with its windowClosing
    //  method, and close events from buttons or other user
    //  components with its actionPerformed method.
    //
    //  The action is to logout and dispose of this window.

    private static class WindowCloser extends WindowAdapter
        implements ActionListener
    {
        Frame myFrame;

        public WindowCloser( Frame frame ) {
            myFrame = frame;
        }

        public void windowClosing (WindowEvent e)
        {
            this.actionPerformed( null );
        }

        public void actionPerformed(ActionEvent e)
        {
            myFrame.dispose();
        }
    }
}
```

```java
// joi/10/juno/InterpreterInterface.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * Juno needs an interpreter to process the user’s response to
 * the login: prompt (or what she enters on a GUILoginConsole).
 *
 * Each Shell needs an interpreter for shell command lines,
 * whether entered with a GUI or a CLI.
 *
 * @version 10
 */

public interface InterpreterInterface
{
    /**
     * Interpret a command line String.
     *
     * @param str the String to interpret
     * @return true, unless str tells you there’s nothing to follow
     */

    public boolean interpret( String str );
}
```

```java
// joi/10/juno/InputInterface.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 *  Juno consoles use the same abstract method
 *  for input, so it is specified here.
 */

public interface InputInterface
{
   /**
    * Read a line (terminated by a newline).
    *
    * @param promptString output string to prompt for input
    * @return the string (without the newline character)
    */

   public String readLine( String promptString );
}
```

```java
// joi/10/juno/OutputInterface.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 *  All Juno consoles use the same abstract methods
 *  for output, so they are specified here.
 */

public interface OutputInterface
{
    /**
     * Write a String followed by a newline
     * to console output location.
     *
     * @param str - the string to write
     */

    public void println(String str );

    /**
     * Write a String followed by a newline
     * to console error output location.
     *
     * @param str - the String to write
     */

    public void errPrintln( String str );

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if echoing input.
     */

    public boolean isEchoInput();

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if GUI
     */

    public boolean isGUI();

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if remote
     */

    public boolean isRemote();
}
```

```java
// joi/10/juno/JunoTerminal.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A Command line interface terminal for Juno.
 *
 * @version 10
 */

public class JunoTerminal
    implements InputInterface, OutputInterface
{
    private Terminal terminal; // the delegate terminal
    private boolean  echo;     // are we echoing input?

    /**
     * Construct a JunoTerminal
     *
     * Allows for input echo, when, for example, input is redirected
     * from a file.
     *
     * @param echo whether or not input should be echoed.
     */

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

    // Implement InputInterface

   /**
    * Read a line (terminated by a newline).
    *
    * @param promptString output string to prompt for input
    * @return the string (without the newline character)
    */

    public String readLine( String promptString )
    {
        return terminal.readLine( promptString );
    }

    // Implement OutputInterface

   /**
     * Write a String followed by a newline
     * to console output location.
     *
     * @param str - the string to write
     */

    public void println(String str )
```

```java
    {
        terminal.println( str );
    }

    /**
     * Write a String followed by a newline
     * to console error output location.
     *
     * @param str - the String to write
     */

    public void errPrintln(String str )
    {
        terminal.errPrintln( str );
    }

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if echoing input.
     */

    public boolean isEchoInput()
    {
        return echo;
    }

    /**
     * Query what kind of console this is.
     *
     * @return false, since it is not a GUI
     */

    public boolean isGUI()
    {
        return false;
    }

    /**
     * Query what kind of console this is.
     *
     * @return false, since it is not remote.
     */

    public boolean isRemote()
    {
        return false;
    }
}
```

```java
// joi/10/juno/RemoteConsole.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.io.*;
import java.net.*;
import java.util.*;
import java.text.*;

/**
 * A remote console listens on a port for a remote login to
 * a running Juno system server.
 *
 * @version 10
 */

public class RemoteConsole extends Thread
    implements OutputInterface, InputInterface
{
    // default just logs connection start and end
    // change to true to log all i/o
    private static boolean logall = false;

    private Juno system;
    private boolean echo;
    private InterpreterInterface interpreter;

    private Socket clientSocket;
    private BufferedReader in;
    private PrintWriter out;
    private int sessionCount = 0;

    private PrintWriter junoLog;

    /**
     * Construct a remote console to listen for users trying
     * to connect to Juno.
     *
     * Called from Juno main.
     *
     * @param system the Juno system setting up this console.
     * @param echo whether or not input should be echoed.
     * @param port the port on which to listen for requests.
     */

    public RemoteConsole( Juno system, boolean echo, int port )
    {
        this.echo = echo;
        Date now  = new Date();
        junoLog   = openlog(now);
        log("*** Juno server started " + now + "\n");
        try {
            ServerSocket ss = new ServerSocket(port);
            while (true) {
                clientSocket = ss.accept();
```

```java
                new RemoteConsole( system, echo, clientSocket,
                                   junoLog, ++sessionCount).start();
            }
        }
        catch (IOException e) {
            System.out.println("Remote login not supported");
            System.exit(0);
        }
        finally {
            system.shutDown();
        }
    }

    /**
     * Construct a remote console for a single remote user.
     *
     * @param system the Juno system to which the user is connecting.
     * @param echo whether or not input should be echoed.
     * @param clientSocket the socket for the user’s connection
     * @param junoLog track all user i/o
     * @param sessionCount this session’s number
     */

    public RemoteConsole( Juno system, boolean echo, Socket clientSocket,
                          PrintWriter junoLog, int sessionCount )
    {
        this.system = system;
        this.echo = echo;
        this.clientSocket = clientSocket;
        this.junoLog = junoLog;
        this.sessionCount = sessionCount;
    }

    /**
     * Action when the thread for this session starts.
     */

    public void run()
    {
        log("*** " + sessionCount + ", " +
            clientSocket.getInetAddress()  + ", "  +
            new Date());
        try {
            setUpConnection();
            String s = this.readLine
                ("Please sign the guest book (name, email): ");
            this.println("Thanks, " + s);
            if (!logall) {
                log("guest book: " + s);
            }
            new LoginInterpreter( system, this ).CLILogin();
            clientSocket.close();
        }
        catch (IOException e) {
            log("*** Error " + e);
        }
        log("*** end session " + sessionCount);
    }

    /**
     * Create the readers and writers for the socket
     * for this session.
     */

    private void setUpConnection()
         throws IOException
    {
        in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
        out = new PrintWriter(
                new OutputStreamWriter(clientSocket.getOutputStream()));
    }

    // implement the InputInterface

   /**
    * Read a line (terminated by a newline) from console socket.
    *
    * Log the input line before returning it if required.
    *
    * @param promptString output string to prompt for input
    * @return the string (without the newline character)
    */

    public String readLine( String promptString )
    {
        String s = "";
        this.print(promptString);
        out.flush();
        try {
            s = in.readLine();
            if (logall) {
                log("> " + s);
            }
            if (echo) {
                out.println(s);
            }
        }
        catch (IOException e) {
            String msg = "IO error reading from remote console";
            System.out.println(msg);
            out.println(msg);
        }
        return s;
    }

    /**
     * Write a String to console socket.
     *
     * Log the output if required.
     *
     * @param str - the string to write
```

```java
     */

    public void print( String str )
    {
        out.print( str );
        out.flush();
        if (logall) {
            log("< " + str + "\\");
        }
    }

    // implement the OutputInterface

    /**
     * Write a String followed by a newline
     * to console socket.
     *
     * Log the output if required.
     *
     * @param str - the string to write
     */

    public void println( String str )
    {
        out.println( str + '\r' );
        out.flush();
        if (logall) {
            log("< " + str);
        }
    }

    /**
     * Write a String followed by a newline
     * to console error output location. That's
     * just the socket.
     *
     * @param str - the String to write
     */

    public void errPrintln(String str )
    {
        println( str );
    }

    /**
     * Query what kind of console this is.
     *
     * @return false since it's not a GUI.
     */

    public boolean isGUI()
    {
        return false;
    }

    /**
```

```
     * Query what kind of console this is.
     *
     * @return true since it is remote.
     */

    public boolean isRemote()
    {
        return true;
    }

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if echoing input.
     */

    public boolean isEchoInput()
    {
        return echo;
    }

    /**
     * Log a String.
     *
     * @param str the String to log.
     */

    private void log(String str)
    {
        junoLog.println(sessionCount + ": " + str);
        junoLog.flush();
    }

    /**
     * Open a log for this console.
     *
     * @param now the current Date.
     */

    private PrintWriter openlog(Date now)
    {
        PrintWriter out = null;
        SimpleDateFormat formatter
            = new SimpleDateFormat ("MMM.dd:hh:mm:ss");
        String dateString = formatter.format(now);
        String filename = "log-" + dateString;
        try { out = new PrintWriter(
                    new BufferedWriter(
                    new FileWriter(filename)));
        }
        catch (Exception e) {
            out = new PrintWriter(new FileWriter(FileDescriptor.out));
        }
        return out;
    }
}
```