```java
// joi/7/juno/Juno.java
//
//
//  Copyright 2003 Bill Campbell and Ethan Bolker

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

/**
 * Juno (Juno’s Unix NOt) mimics a command line operating system
 * like Unix.
 * <p>
 * A Juno system has a name, a set of Users, a JFile system,
 * a login process and a set of shell commands.
 *
 * @see User
 * @see JFile
 * @see ShellCommand
 *
 * @version 7
 */

public class Juno
{
    private final static String OS      = "Juno";
    private final static String VERSION = "7";

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

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

    private ShellCommandTable commandTable; // shell commands

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

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

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

        // the file system

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

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

        // create, then start a command line login interpreter

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

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

    public String getHostName()
    {
        return hostName;
    }

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

    public String getOS()
    {
        return OS;
    }

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

    public String getVersion()
    {
        return VERSION;
    }

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

    public Directory getUserHomes()
    {
        return userHomes;
    }

    /**
     * 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 realName her real name.
     * @return newly created User.
     */

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

    /**
     * The Juno system may be given the following command line
     * arguments.
     * <pre>
     *
     * -e:         Echo all input (useful for testing).
     *
     * -version:   Report the version number and exit.
     *
     * [hostname]: The name of the host on which
     *             Juno is running (optional).
     * </pre>
     */

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

```java
    {
        // Parse command line options

        boolean echoInput = false;
        String  hostName  = "mars";

        for (int i=0; i < args.length; i++) {
            if (args[i].equals("-version")) {
                System.out.println( OS + " version " + VERSION );
                System.exit(0);
            }
            if (args[i].equals("-e")) {
                echoInput = true;
            }
            else {
                hostName = args[i];
            }
        }

        // create a Juno instance, which will start itself

        new Juno( hostName, echoInput );
    }
}
```

```java
// joi/7/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 7
 */

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

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

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

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

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

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

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

    public void CLILogin()
    {
```

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

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

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

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

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

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

    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/7/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 7
 */

public class Shell
{
    private Juno system;       // the operating system object
    private User user;         // the user logged in
    private Terminal 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.
     */

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

    // Run the command line interpreter

    private void CLIShell()
    {
        boolean moreWork = true;
        while(moreWork) {
            moreWork = interpret( console.readLine( getPrompt() ) );
        }
        console.println("goodbye");
    }

    // Interpret a String of the form
    //     shellcommand command-arguments
```

```java
    //
    // return true, unless shell command is logout.

    private 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 ) {                             // EEE
            console.errPrintln("Unknown command: " + commandName);// EEE
            return true;
        }                                                        // EEE
        try {                                                    // EEE
            commandObject.doIt( st, this );                      // EEE
        }                                                        // EEE
        catch (ExitShellException e) {
            return false;
        }
        catch (BadShellCommandException e) {                      // EEE
            console.errPrintln( "Usage: " + commandName + " " +  // EEE
                                e.getCommand().getArgString() ); // EEE
        }                                                         // EEE
        catch (JunoException e) {                                 // EEE
            console.errPrintln( e.getMessage() );                 // EEE
        }                                                         // EEE
        catch (Exception e) {                                     // EEE
            console.errPrintln( "you should never get here" );   // EEE
            console.errPrintln( e.toString() );                   // EEE
        }                                                         // EEE
        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() + "> ";
    }

    /**
     * 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 Terminal 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/7/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 7
 */

public abstract class ShellCommand
{
    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
     */

    public abstract void doIt( StringTokenizer args, Shell sh )
```

```java
        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/7/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 7
 */

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

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

    public ShellCommandTable()
    {
        fillTable();
    }

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

    public ShellCommand lookup( String key )
    {
        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 {
            commandObject =
```

```java
                    (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/7/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 7
 */

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/7/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 7
 */

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

```java
        }                                                                                  // EEE
    }
}
```

```java
// joi/7/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 7
 */

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/7/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 7
 */

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 );
    }
}
```

```
// joi/7/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 7
 */

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
// joi/7/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 7
 */

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
    {
        Terminal 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/7/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 7
 */

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/7/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 7
 */

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/7/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 7
 */

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

    public static final String separator = File.separator;

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

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

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

```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;
    }
```

```

    /**
     * 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/7/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 7
 */

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/7/juno/TextFile.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * A TextFile is a JFile that holds text.
 *
 * @version 7
 */

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);
    }
}
```

```java
// joi/7/juno/User.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * Model a Juno user.  Each User has a login name,
 * a home directory, and a real name.
 *
 * @version 7
 */

public class User
{
    private String name;     // the User’s login name
    private Directory home;  // her home Directory
    private String realName; // her real name

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

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

    /**
     * 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();
    }
```

```java

    /**
     * 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/7/juno/JunoException.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A general Juno Exception.
 *
 * @version 7
 */

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/7/juno/BadShellCommandException.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * The Exception generated when a ShellCommand is misused.
 *
 * @version 7
 */

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/7/juno/ExitShellException.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * Exception raised for exiting a shell.
 *
 * @version 7
 */

public class ExitShellException extends JunoException
{
}
```