

```

1 // foj/l/bank/BankAccount.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A BankAccount object has a private field to keep track
8  * of this account's current balance, and public methods to
9  * return and change the balance.
10 *
11 * @see Bank
12 * @version 1
13 */
14
15 public class BankAccount
16 {
17     private int balance; // work only in whole dollars
18
19     /**
20      * A constructor for creating a new bank account.
21      *
22      * @param initialBalance the opening balance.
23      */
24
25     public BankAccount( int initialBalance )
26     {
27         this.deposit( initialBalance );
28     }
29
30     /**
31      * Withdraw the amount requested.
32      *
33      * @param amount the amount to be withdrawn.
34      */
35
36     public void withdraw( int amount )
37     {
38         balance = balance - amount;
39     }
40
41     /**
42      * Deposit the amount requested.
43      *
44      * @param amount the amount to be deposited.
45      */
46
47     public void deposit( int amount )
48     {
49         balance = balance + amount;
50     }
51
52     /**
53      * The current account balance.
54      *
55      * @return the current balance.
56      */

```

```

57
58     public int getBalance()
59     {
60         return balance;
61     }
62 }

```

```

1 // fo1/1/bank/Bank.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A Bank object simulates the behavior of a simple bank/ATM.
8  * It contains a Terminal object and two BankAccount objects.
9
10 * Its single public method is open, which opens this Bank
11 * for business, prompting the customer for input.
12 *
13 * To create a Bank and open it for business issue the command
14 * <code>java Bank</code>.
15 *
16 * @see BankAccount
17 * @version 1
18 */
19
20 public class Bank
21 {
22     private String bankName; // the name of this Bank
23
24     private Terminal atm; // for talking with the customer
25
26     private BankAccount account1; // two accounts to play with
27     private BankAccount account2;
28
29     private static final int INITIAL_BALANCE = 200;
30     private static final String HELPSTRING =
31         "Transactions: exit, help, deposit, withdraw, balance";
32
33     /**
34      * Construct a Bank with the given name.
35      * Create two new BankAccounts, each with a starting balance
36      * of InitialBalance.
37      *
38      * @param name the name of the Bank.
39      */
40
41     public Bank( String name )
42     {
43         bankName = name;
44         atm = new Terminal();
45         account1 = new BankAccount( INITIAL_BALANCE );
46         account2 = new BankAccount( INITIAL_BALANCE );
47     }
48
49     /**
50      * Open the Bank for business.
51
52      * Send a whichAccount message prompting for a BankAccount
53      * number, then send a processTransactionsForAccount
54      * message to do the work.
55      */

```

```

57     public void open()
58     {
59         atm.println( "Welcome to " + bankName );
60         boolean bankIsOpen = true;
61         while ( bankIsOpen ) {
62             BankAccount account = this.whichAccount();
63             if ( account == null ) {
64                 bankIsOpen = false;
65             }
66             else {
67                 this.processTransactionsForAccount(account);
68             }
69         }
70         atm.println( "Goodbye from " + bankName );
71     }
72
73     // Prompt the user for an account number and return the
74     // corresponding BankAccount object. Return null when
75     // the Bank is about to close.
76
77     private BankAccount whichAccount()
78     {
79         int accountNumber =
80             atm.readInt( "Account number ( 1 or 2 ), 0 to shut down: " );
81
82         if ( accountNumber == 1 ) {
83             return account1;
84         }
85         else if ( accountNumber == 2 ) {
86             return account2;
87         }
88         else if ( accountNumber == 0 ) {
89             return null;
90         }
91         else {
92             atm.println( "No account numbered " +
93                 accountNumber + "; try again" );
94             return this.whichAccount();
95         }
96     }
97
98     // Prompt the user for transaction to process.
99     // Then send an appropriate message to account.
100
101     private void processTransactionsForAccount( BankAccount account )
102     {
103         atm.println( HELPSTRING );
104
105         boolean moreTransactions = true;
106         while ( moreTransactions ) {
107             String command = atm.readWord( "transaction: " );
108             if ( command.equals( "exit" ) ) {
109                 moreTransactions = false;
110             }
111             else if ( command.equals( "help" ) ) {
112                 atm.println( HELPSTRING );

```

```
113     }
114     else if ( command.equals( "deposit" ) ) {
115         int amount = atm.readInt( "amount: " );
116         account.deposit( amount );
117     }
118     else if ( command.equals( "withdraw" ) ) {
119         int amount = atm.readInt( "amount: " );
120         account.withdraw( amount );
121     }
122     else if ( command.equals( "balance" ) ) {
123         atm.println( account.getBalance() );
124     }
125     else{
126         atm.println("sorry, unknown transaction" );
127     }
128 }
129 }
130 }
131 }
132 /**
133  * The Bank simulation program begins here when the user
134  * issues the command <code>java Bank</code>.
135  * @param args the command line arguments (ignored).
136  */
137 public static void main( String[] args )
138 {
139     Bank javaBank = new Bank( "Engulf and Devour" );
140     javaBank.open();
141 }
142 }
143 }
```

```

1 // fo1/1/lights/TrafficLight.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.awt.*;
7 import java.awt.event.*;
8
9 /**
10  * A TrafficLight has three lenses: red, yellow and green.
11  * It can be set to signal Go, Caution, Stop or Walk.
12  */
13 * @version 1
14 */
15
16 public class TrafficLight extends Panel
17 {
18     // Three Lenses and a Button
19
20     private Lens red      = new Lens( Color.red );
21     private Lens yellow   = new Lens( Color.yellow );
22     private Lens green    = new Lens( Color.green );
23     private Button nextButton = new Button("Next");
24
25     /**
26      * Construct a traffic light.
27      */
28
29     public TrafficLight()
30     {
31         this.setLayout(new BorderLayout());
32
33         // create a Panel for the Lenses
34         Panel lensPanel = new Panel();
35         lensPanel.setLayout( new GridLayout( 3, 1 ) );
36         lensPanel.add( red );
37         lensPanel.add( yellow );
38         lensPanel.add( green );
39         this.add( BorderLayout.NORTH, lensPanel );
40
41         // configure the "Next" button
42         Sequencer sequencer = new Sequencer( this );
43         NextButtonListener payAttention =
44             new NextButtonListener( sequencer );
45         nextButton.addActionListener( payAttention );
46         this.add( BorderLayout.CENTER, nextButton);
47     }
48
49     // Methods that change the light
50
51     /**
52      * Set the light to stop (red).
53      */
54
55     public void setStop()
56     {

```

```

57         red.turnOn();
58         yellow.turnOff();
59         green.turnOff();
60     }
61
62     /**
63      * Set the light to caution (yellow).
64      */
65
66     public void setCaution()
67     {
68         red.turnOff();
69         yellow.turnOn();
70         green.turnOff();
71     }
72
73     /**
74      * Set the light to go (green).
75      */
76
77     public void setGo()
78     {
79         red.turnOff();
80         yellow.turnOff();
81         green.turnOn();
82     }
83
84     /**
85      * Set the light to walk.
86      *
87      * (In Boston, red and yellow signal walk.)
88      */
89
90     public void setWalk()
91     {
92         red.turnOn();
93         yellow.turnOn();
94         green.turnOff();
95     }
96
97     /**
98      * The traffic light simulation starts at main.
99      */
100     * @param args ignored.
101     */
102
103     public static void main( String[] args )
104     {
105         JFrame frame
106             = new JFrame();
107         TrafficLight light = new TrafficLight();
108         frame.addWindowListener( new ShutdownLight() );
109         frame.pack();
110         frame.show();
111     }
112

```

```
113 // A Shutdownlight instance handles close events generated
114 // by the underlying window system with its windowClosing
115 // method.
116 //
117 // This is an inner class, declared inside the
118 // TrafficLight class since it's used only here.
119
120 private static class ShutdownLight extends WindowAdaptrer
121 {
122     // Close the window by shutting down the light.
123     public void windowClosing (WindowEvent e)
124     {
125         System.exit(0);
126     }
127 }
128 }
129 }
130 }
131 }
```

```
1 // foj/l/lights/NextButtonListener.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.awt.event.*;
7
8 /**
9  * A NextButtonListener sends a "next" message to its
10  * Sequencer each time a button to which it is listening
11  * is pressed.
12  *
13  * @version 1
14  */
15
16 public class NextButtonListener implements ActionListener
17 {
18     private Sequencer sequencer;
19
20     /**
21      * Construct a listener that "listens for" a user's
22      * pressing the "Next" button.
23      *
24      * @param sequencer the Sequencer for the TrafficLight.
25      */
26
27     public NextButtonListener( Sequencer sequencer )
28     {
29         this.sequencer = sequencer;
30     }
31
32     /**
33      * The action performed when a push of the button is detected:
34      * send a next message to the Sequencer to advance it to
35      * its next state.
36      *
37      * @param event the event detected at the button.
38      */
39
40     public void actionPerformed( ActionEvent event )
41     {
42         this.sequencer.next();
43     }
44 }
```

```

1 // fo1/1/lights/Sequencer.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A Sequencer controls a TrafficLight. It maintains fields
8  * for the light itself and the current state of the light.
9
10 * Each time it receives a "next" message, it advances to the
11 * next state and sends the light an appropriate message.
12 *
13 * @version 1
14 */
15
16 public class Sequencer
17 {
18     // the TrafficLight this Sequencer controls
19     private TrafficLight light;
20
21     // represent the states by ints
22     private final static int GO      = 0;
23     private final static int CAUTION = 1;
24     private final static int STOP    = 2;
25
26     private int currentState;
27
28     /**
29      * Construct a sequencer to control a TrafficLight.
30      *
31      * @param light the TrafficLight we wish to control.
32      */
33
34     public Sequencer( TrafficLight light )
35     {
36         this.light = light;
37         this.currentState = GO;
38         this.light.setGo();
39     }
40
41     /**
42      * How the light changes when a next Button is pressed
43      * depends on the current state. The sequence is
44      * GO -> CAUTION -> STOP -> GO.
45      */
46
47     public void next()
48     {
49         switch ( currentState ) {
50
51             case GO:
52                 this.currentState = CAUTION;
53                 this.light.setCaution();
54                 break;
55
56             case CAUTION:

```

```

57         this.currentState = STOP;
58         this.light.setStop();
59         break;
60
61         case STOP:
62             this.currentState = GO;
63             this.light.setGo();
64             break;
65
66         default: // This will never happen
67             System.err.println("What color is the light?!");
68         }
69     }
70 }

```

```

1 // fo1/1/lights/Lens.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.awt.*;
7
8 /**
9  * A Lens has a certain color and can either be turned on
10 * (the color) or turned off (black).
11 *
12 * @version 1
13 */
14
15 public class Lens extends Canvas
16 {
17     private Color onColor; // color on
18     private Color offColor = Color.black; // color off
19     private Color currentColor; // color the lens is now
20
21     private final static int SIZE = 100; // how big is this Lens?
22     private final static int OFFSET = 20; // offset of Lens in Canvas
23
24     /**
25      * Construct a Lens to display a given color.
26      *
27      * The lens is black when it's turned off.
28      *
29      * @param color the color of the lens when it is turned on.
30      */
31
32     public Lens( Color color )
33     {
34         this.setBackground( Color.black );
35         this.onColor = color;
36         this.setSize( SIZE , SIZE );
37         this.turnOff();
38     }
39
40     /**
41      * How this Lens paints itself.
42      *
43      * @param g a Graphics object to manage brush and color information.
44      */
45
46     public void paint( Graphics g )
47     {
48         g.setColor( this.currentColor );
49         g.fillRect( OFFSET, OFFSET,
50                   SIZE - OFFSET*2, SIZE - OFFSET*2 );
51     }
52
53     /**
54      * Have this Lens display its color.
55      */
56

```

```

57     public void turnOn()
58     {
59         currentColor = onColor;
60         this.repaint();
61     }
62
63     /**
64      * Darken this lens.
65      */
66
67     public void turnOff()
68     {
69         currentColor = offColor;
70         this.repaint();
71     }
72 }

```



```

1 // foj/1/estore/ESTore.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * An EStore object simulates the behavior of a simple on line
8  * shopping web site.
9
10 * It contains a Terminal object to model the customer's browser
11 * and several Item objects a customer can add to her ShoppingCart.
12
13 * @version 1
14 */
15
16 public class EStore
17 {
18     private String storeName = "Virtual Minimal Minimal";
19
20     // Use a Terminal object to communicate with customers.
21     private Terminal browser = new Terminal();
22
23     // The store stocks two kinds of Items.
24     private Item widget = new Item(10); // widgets cost $10
25     private Item gadget = new Item(13); // gadgets cost $13
26
27     private String selectionList = "(gadget, widget, checkout)";
28
29     /**
30      * Visit this EStore.
31
32      * Loop allowing visitor to select items to add to her
33      * ShoppingCart.
34      */
35
36     public void visit()
37     {
38         // Create a new, empty ShoppingCart.
39         ShoppingCart basket = new ShoppingCart();
40
41         // Print a friendly welcome message.
42         browser.println("Welcome to " + storeName );
43
44         // Change to false when customer is ready to leave:
45         boolean stillShopping = true;
46
47         while ( stillShopping ) {
48             Item nextPurchase = selectItem();
49             if ( nextPurchase == null ) {
50                 stillShopping = false;
51             }
52             else {
53                 basket.add( nextPurchase );
54             }
55         }
56         int numberPurchased = basket.getCount();

```

```

57     int totalCost      = basket.getCost();
58     browser.println("We are shipping " + numberPurchased + " Items");
59     browser.println("and charging your account $" + totalCost);
60     browser.println("Thank you for shopping at " + storeName);
61 }
62
63 // Discover what the customer wants to do next:
64 // send browser a message to get customer input
65 // examine response to make a choice
66 // If response makes no sense give customer another chance
67
68     private Item selectItem()
69     {
70         String itemName =
71             browser.readWord("Item " + selectionList + " :");
72
73         if ( itemName.equals("widget") ) {
74             return widget;
75         }
76         else if ( itemName.equals("gadget") ) {
77             return gadget;
78         }
79         else if ( itemName.equals("checkout") ) {
80             return null;
81         }
82         else {
83             browser.println("No item named " +
84                 itemName + "; try again" );
85             return selectItem(); // try again
86         }
87     }
88
89     /**
90      * The EStore simulation program begins here when the user
91      * issues the command <code>java EStore</code>.
92      */
93
94     public static void main( String[] args )
95     {
96         // Print this to simulate delay while browser finds store
97         System.out.println("connecting ...");
98
99         // Create the EStore object.
100        EStore website = new EStore();
101
102        // Visit it.
103        website.visit();
104    } // end of class EStore
105

```

```
1 // fo1/l/estore/Item.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * An Item models an object that might be stocked in a store.
8  * Each Item has a cost.
9  *
10 * @version 1
11 */
12
13 public class Item
14 {
15     private int cost;
16
17     /**
18      * Construct an Item object.
19      *
20      * @param itemCost the cost of this Item.
21      */
22
23     public Item( int itemCost )
24     {
25         cost = itemCost;
26     }
27
28     /**
29      * How much does this Item cost?
30      *
31      * @return the cost.
32      */
33
34     public int getCost()
35     {
36         return cost;
37     }
38 }
```

```

1 // fo1/l/estore/ShoppingCart.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A ShoppingCart keeps track of a customer's purchases.
8  *
9  * @see EStore
10 * @version 1
11 */
12
13 public class ShoppingCart
14 {
15     private int count; // number of Items in this ShoppingCart
16     private int cost; // cost of Items in this ShoppingCart
17
18     /**
19      * Construct a new empty ShoppingCart.
20      */
21
22     public ShoppingCart()
23     {
24         count = 0;
25         cost = 0;
26     }
27
28     /**
29      * When this ShoppingCart is asked to add an Item to itself
30      * it updates its count field and then updates its cost
31      * field by sending the Item a getCost message.
32      *
33      * @param purchase the Item being added to this ShoppingCart.
34      */
35
36     public void add( Item purchase )
37     {
38         count++; // Java idiom for count = count + 1;
39         cost = cost + purchase.getCost();
40     }
41
42     /**
43      * What happens when this ShoppingCart is asked how many
44      * Items it contains.
45      *
46      * @return the count of Items.
47      */
48
49     public int getCount()
50     {
51         return count;
52     }
53
54     /**
55      * What happens when this ShoppingCart is asked the total
56      * cost of the Items it contains.

```

```

57     *
58     * @return the total cost.
59     */
60     public int getCost()
61     {
62         return cost;
63     }
64 }
65 }

```

```
1 // fo1/2/change/Change.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * Program to make change.
8  * Uses the Terminal method readInt() for prompted integer input.
9  *
10 * @version 2
11 */
12
13 public class Change
14 {
15     /**
16      * Illustrate simple arithmetic.
17      */
18
19     public static void main (String[] args)
20     {
21         Terminal terminal = new Terminal();
22         int amount;
23
24         amount = terminal.readInt("Amount, in cents: ");
25         int dimes = amount/10;
26         amount = amount % 10;
27         int nickels = amount / 5;
28         amount = amount % 5;
29         terminal.println(dimes + " dimes");
30         terminal.println(nickels + " nickels");
31         terminal.println(amount + " pennies");
32     }
33 }
```

```

1 // jol/2/linear/Temperatures.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * Temperature conversion program,
8  * for exercising LinearEquation objects.
9  *
10 * @version 2
11 */
12
13 public class Temperatures
14 {
15     /**
16      * First a hardcoded test of Celsius-Fahrenheit conversion,
17      * then a loop allowing the user to test interactively.
18      */
19
20     public static void main( String[] args )
21     {
22         Terminal terminal = new Terminal();
23
24         // create a Celsius to Fahrenheit converter
25         LinearEquation c2f = new LinearEquation( 9.0/5.0, 32.0 );
26
27         // ask it to tell us its inverse, for F to C
28         LinearEquation f2c = c2f.getInverse();
29
30         ////////////////////////////////////////////////////////////////////
31         // Testing style 1: Hard coded, self-documenting //
32         ////////////////////////////////////////////////////////////////////
33
34         terminal.println( "Hard coded self documenting tests:" );
35         terminal.print( "c2f.compute( 0.0 ), should see 32.0: " );
36         terminal.println( c2f.compute( 0.0 ) );
37         terminal.print( "f2c.compute( 212.0 ), should see 100.0: " );
38         terminal.println( f2c.compute( 212.0 ) );
39
40         ////////////////////////////////////////////////////////////////////
41         // Testing style 2: Interactive //
42         ////////////////////////////////////////////////////////////////////
43
44         terminal.println();
45         terminal.println( "Interactive tests:" );
46         while ( terminal.readYesOrNo( "more?" ) ) {
47             double degreesCelsius =
48                 terminal.readDouble( "Celsius: " );
49             terminal.println( " = "
50                 + c2f.compute( degreesCelsius )
51                 + " degrees Fahrenheit" );
52             double degreesFahrenheit =
53                 terminal.readDouble( "degrees Fahrenheit: " );
54             terminal.println( " = "
55                 + f2c.compute( degreesFahrenheit )
56                 + " degrees Celsius" );

```

```

57     }
58 }
59 }

```

```

1 // fo1/2/LinearEquation.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5 /**
6  * A LinearEquation models equations of the form  $y = mx + b$ .
7  *
8  * @version 2
9  */
10
11 public class LinearEquation
12 {
13     private double m; // The equations's slope
14     private double b; // The equations's y-intercept
15
16     /**
17      * Construct a LinearEquation from a slope and y-intercept.
18      *
19      * @param m the slope.
20      * @param b the y-intercept.
21      */
22
23     public LinearEquation( double m, double b )
24     {
25         this.m = m;
26         this.b = b;
27     }
28
29     /**
30      * Construct a LinearEquation from two points.
31      *
32      * @param x1 the x coordinate of the first point
33      * @param y1 the y coordinate of the first point
34      * @param x2 the x coordinate of the second point
35      * @param y2 the y coordinate of the second point
36      */
37
38     public LinearEquation( double x1, double y1,
39                           double x2, double y2 )
40     {
41         m = (y2 - y1) / (x2 - x1);
42         b = y1 - x1 * m;
43     }
44
45     /**
46      * Compute Y, given x.
47      *
48      * @param x the input value.
49      * @return the corresponding value of y: mx+b.
50      */
51
52     public double compute( double x )
53     {
54         return m*x + b;
55     }
56

```

```

57
58     /**
59      * Compute the inverse of this linear equation.
60      *
61      * @return the LinearEquation object you get by "solving for x".
62      */
63
64     public LinearEquation getInverse()
65     {
66         return new LinearEquation( 1.0/m, -b/m );
67     }
68 }

```

```
1 // fo1/2/arithmetic/InArithmetic.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * Interactive play with integer arithmetic in Java,
8  * using a Terminal for input and output.
9  */
10
11 public class InArithmetic
12 {
13     private static Terminal terminal = new Terminal();
14
15     /**
16      * main prompts for pairs of numbers to add and to divide
17      * until the bored user decides to quit.
18      */
19
20     public static void main(String[] args)
21     {
22         while ( terminal.readYesOrNo( "Try int z = x + y ? " ) ) {
23             tryIntegerAddition( );
24         }
25         while ( terminal.readYesOrNo( "Try int z = x / y ? " ) ) {
26             tryIntegerDivision( );
27         }
28     }
29
30     // Prompt for two ints and add them.
31
32     private static void tryIntegerAddition()
33     {
34         int x = terminal.readInt( "x = " );
35         int y = terminal.readInt( "y = " );
36         terminal.println( "z = " + (x+y) );
37     }
38
39     // Prompt for two ints and divide the first by
40     // the second.
41
42     private static void tryIntegerDivision()
43     {
44         int x = terminal.readInt( "x = " );
45         int y = terminal.readInt( "y = " );
46         terminal.println( "z = " + (x/y) );
47     }
48 }
```

```

1 // jol/3/textfiles/TextFile.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.util.Date;
7
8 /**
9  * A TextFile mimics the sort of text file that one finds
10 * on a computer's file system. It has an owner,
11 * a create date (when the file was created),
12 * a modification date (when the file was last modified),
13 * and String contents.
14
15 * @version 3
16 */
17
18 public class TextFile
19 {
20     // Private Implementation
21
22     private String owner; // Who owns the file.
23     private Date createdAt; // When the file was created.
24     private Date modDate; // When the file was last modified.
25     private String contents; // The text stored in the file.
26
27     // Public Interface
28
29     /**
30      * Construct a new TextFile with given owner and
31      * contents; set the creation and modification dates.
32      *
33      * @param owner the user who owns the file.
34      * @param contents the file's initial contents.
35      */
36
37     public TextFile( String owner, String contents )
38     {
39         this.owner = owner;
40         this.contents = contents;
41         createdAt = new Date(); // date and time now
42         modDate = createdAt;
43     }
44
45     /**
46      * Replace the contents of the file.
47      *
48      * @param contents the new contents.
49      */
50
51     public void setContents( String contents )
52     {
53         this.contents = contents;
54         modDate = new Date();
55     }
56

```

```

57     /**
58      * The contents of a file.
59      *
60      * @return String contents of the file.
61      */
62
63     public String getContents()
64     {
65         return contents;
66     }
67
68     /**
69      * Append text to the end of the file.
70      *
71      * @param text the text to be appended.
72      */
73
74     public void append( String text )
75     {
76         this.setContents( contents + text );
77     }
78
79     /**
80      * Append a new line of text to the end of the file.
81      *
82      * @param text the text to be appended.
83      */
84
85     public void appendline( String text )
86     {
87         this.setContents( contents + '\n' + text );
88     }
89
90     /**
91      * The size of a file.
92      *
93      * @return the integer size of the file
94      * (the number of characters in its String contents)
95      */
96
97     public int getSize()
98     {
99         int charCount;
100         charCount = contents.length();
101         return charCount;
102     }
103
104     /**
105      * The data and time of the file's creation.
106      *
107      * @return the file's creation date and time.
108      */
109
110     public String getCreateDate()
111     {
112         return createdAt.toString();
113     }

```



```

113     }
114
115     /**
116     * The date and time of the file's last modification.
117     *
118     * @return the date and time of the file's last modification.
119     */
120
121     public String getModDate()
122     {
123         return modDate.toString();
124     }
125
126     /**
127     * The file's owner.
128     *
129     * @return the owner of the file.
130     */
131
132     public String getOwner()
133     {
134         return owner;
135     }
136
137     /**
138     * A definition of main(), used only for testing this class.
139     *
140     * Executing
141     * <pre>
142     * %> java TextFile
143     * </pre>
144     * produces the output:
145     * <pre>
146     * TextFile myTextFile contains 13 characters.
147     * Created by Bill, Sat Dec 29 14:02:37 EST 2001
148     * Hello, world.
149     *
150     * append new line "How are you today?"
151     * Hello, world.
152     * How are you today?
153     * TextFile myTextFile contains 32 characters.
154     * Modified Sat Dec 29 14:02:38 EST 2001
155     * </pre>
156     */
157
158     public static void main( String[] args )
159     {
160         Terminal terminal = new Terminal();
161         TextFile myTextFile
162             = new TextFile( "bill", "Hello, world." );
163         terminal.println( "TextFile myTextFile contains " +
164             myTextFile.getSize() +
165             " characters." );
166         terminal.println( "Created by " +
167             myTextFile.getOwner() + ", " +
168             myTextFile.getCreatedDate() );

```

```

169         terminal.println( myTextFile.getContents() );
170         terminal.println();
171         terminal.println(
172             "append new line \"How are you today?\"");
173         myTextFile.appendLine( "How are you today?" );
174         terminal.println( myTextFile.getContents() );
175         terminal.println( "TextFile myTextFile contains " +
176             myTextFile.getSize() +
177             " characters." );
178         terminal.println( "Modified " +
179             myTextFile.getModDate() );
180     }
181 }

```

```
1 // fo1/3/shapes/DemoShapes.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A short demonstration program for HLine and Box.
8  *
9  * @version 3
10 */
11
12 public class DemoShapes
13 {
14     /**
15      * Paint some shapes on a Screen and draw it to a Terminal.
16      */
17
18     public static void main( String[] args )
19     {
20         Terminal t = new Terminal();
21         Screen s = new Screen( 36, 12 );
22
23         HLine h1 = new HLine( 10, 'R' );
24         Box b1 = new Box( 5, 6, 'G' );
25         Box b2 = new Box( 5, 6, 'B' );
26
27         h1.paintOn( s ); // at position (0,0)
28         b1.paintOn( s, 2, 2 );
29         b2.paintOn( s, 4, 5 );
30
31         t.println( "A Screen with an HLine and two Boxes:" );
32         s.draw( t );
33     }
34 }
```

```

1 // jol/3/shapes/HLine.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A horizontal line has a length and a paintChar used
8  * used to paint the line on a Screen.
9  *
10 * @version 3
11 */
12
13 public class HLine
14 {
15     private int length; // length in (character) pixels.
16     private char paintChar; // character used for painting.
17
18     /**
19      * Construct an HLine.
20      *
21      * @param length length in (character) pixels.
22      * @param paintChar character used for painting this line.
23      */
24
25     public HLine( int length, char paintChar )
26     {
27         this.length = length;
28         this.paintChar = paintChar;
29     }
30
31     /**
32      * Paint this HLine on Screen s at position (x,y).
33      *
34      * @param s the Screen on which this line is to be painted.
35      * @param x the x position for the line.
36      * @param y the y position for the line.
37      */
38
39     public void paintOn( Screen s, int x, int y )
40     {
41         for ( int i = 0; i < length; i = i+1 ) {
42             s.paintAt( paintChar, x+1, y );
43         }
44     }
45
46     /**
47      * Paint this HLine on Screen s at position (0,0).
48      *
49      * @param s the Screen on which this line is to be painted.
50      */
51
52     public void paintOn( Screen s )
53     {
54         paintOn( s, 0, 0 );
55     }
56

```

```

57     /**
58      * Get the length of this line.
59      *
60      * @return the length in (character) pixels.
61      */
62
63     public int getLength()
64     {
65         return length;
66     }
67
68     /**
69      * Set the length of this line.
70      *
71      * @param length the new length in (character) pixels.
72      */
73
74     public void setLength( int length )
75     {
76         this.length = length;
77     }
78
79     /**
80      * Unit test for class HLine,
81      * assuming Screen and Terminal work.
82      */
83
84     public static void main( String[] args )
85     {
86         Terminal terminal = new Terminal();
87
88         terminal.println( "Unit test of HLine." );
89         terminal.println( "You should see this Screen twice: " );
90         terminal.println( "++++++");
91         terminal.println( "xxxxxxxxxxx");
92         terminal.println( "xxxxxx");
93         terminal.println( "++++");
94         terminal.println( "++++");
95         terminal.println( "++++");
96         terminal.println( "++++");
97         terminal.println( "++++");
98         terminal.println( "++++");
99
100         Screen screen = new Screen( 20, 6 );
101
102         HLine hline1 = new HLine( 10, 'x' );
103         HLine hline2 = new HLine( 5, '*' );
104         HLine hline3 = new HLine( 1, '1' );
105
106         hline1.paintOn( screen );
107         hline1.setLength(5);
108         hline1.paintOn( screen, 0, 1 );
109         hline2.paintOn( screen, 3, 3 );
110         hline3.paintOn( screen, 4, 4 );
111
112         screen.draw( terminal );

```

```
113 }  
114 }
```

```

1 // fo1/3/shapes/Box.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A Box has a width, a height and a paintChar used
8  * used to paint the Box on a Screen.
9
10 * Examples:
11 * <pre>
12 * new Box( 3, 4, 'G' ) new Box( 1, 1, '$' )
13 *
14 *      GGG           $
15 *      GGG           GGG
16 *      GGG           GGG
17 *
18 * </pre>
19 *
20 * @version 3
21 */
22
23 public class Box
24 {
25     private int width; // width in (character) pixels
26     private int height; // height in (character) pixels
27     private char paintChar; // character used for painting
28
29     /**
30      * Construct a box.
31      *
32      * @param width width in (character) pixels.
33      * @param height height in (character) pixels.
34      * @param paintChar character used for painting this Box.
35      */
36
37     public Box( int width, int height, char paintChar )
38     {
39         this.width = width;
40         this.height = height;
41         this.paintChar = paintChar;
42     }
43
44     /**
45      * Paint this Box on Screen s at position (x,y).
46      *
47      * @param s the screen on which this box is to be painted.
48      * @param x the x position for the box.
49      * @param y the y position for the box.
50      */
51
52     public void paintOn( Screen s, int x, int y )
53     {
54         HLine hline = new HLine( width, paintChar );
55         for ( int i = 0; i < height; i++ ) {
56             hline.paintOn( s, x, y+i );

```

```

57     }
58     }
59
60     /**
61      * Paint this Box on Screen s at position (0,0).
62      *
63      * @param s the Screen on which this box is to be painted.
64      */
65
66     public void paintOn( Screen s )
67     {
68         paintOn( s, 0, 0 ); // or this.paintOn(s,0,0);
69     }
70
71     /**
72      * Get the width of this Box.
73      *
74      * @return width of box (expressed as a number
75      * of characters).
76      */
77
78     public int getWidth()
79     {
80         return width;
81     }
82
83     /**
84      * Get the height of this Box.
85      *
86      * @return the height in (character) pixels.
87      */
88
89     public int getHeight()
90     {
91         return height;
92     }
93
94     /**
95      * Set the width of this Box.
96      *
97      * @param width the new width in (character) pixels.
98      */
99
100     public void setWidth( int width )
101     {
102         this.width = width;
103     }
104
105     /**
106      * Set the height of this Box.
107      *
108      * @param height the new height in (character) pixels.
109      */
110
111     public void setHeight( int height )
112     {

```

```
113     this.height = height;
114 }
115
116 /**
117  * Unit test for class Box,
118  * assuming Screen and Terminal work.
119  */
120
121 public static void main( String[] args )
122 {
123     Terminal terminal = new Terminal();
124
125     terminal.println( "Unit test of Box." );
126     terminal.println( "You should see this Screen twice: " );
127     terminal.println( "++++++");
128     terminal.println( "+RRRR +");
129     terminal.println( "+RRR +");
130     terminal.println( "+RRGGG +");
131     terminal.println( "+RRGGG +");
132     terminal.println( "+RRGGG +");
133     terminal.println( "+ GRRRRRRR +");
134     terminal.println( "++++++");
135     terminal.println();
136
137     Screen screen = new Screen( 20, 6 );
138
139     Box box1 = new Box( 4, 5, 'R' );
140     Box box2 = new Box( 3, 4, 'G' );
141
142     box1.paintOn( screen );
143     box2.paintOn( screen, 2, 2 );
144
145     // test reference model for objects
146     box2 = box1;
147     int oldWidth = box2.getWidth();
148     box1.setWidth( oldWidth+3 );
149     box2.paintOn( screen, 4, 5 );
150
151     screen.draw( terminal );
152 }
153 }
```

```
1 // fo1/3/shapes/TestShapes.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A program to test shapes.
8  *
9  * @version 3
10 */
11
12 class TestShapes
13 {
14     /**
15      * Paint shapes on a Screen and draw it to a Terminal.
16      */
17
18     public static void main( String[] argv )
19     {
20         Terminal t = new Terminal();
21         Screen s;
22
23         t.println( "An empty 10 x 3 Screen:" );
24         s = new Screen( 10, 3 );
25         s.draw( t );
26
27         t.println( "A 20 x 10 Screen with 3 HLines:" );
28         s = new Screen( 20, 10 );
29         HLine h1 = new HLine( 10, 'R' );
30         HLine h2 = new HLine( 15, 'G' );
31
32         h1.paintOn( s, 0, 0 );
33         h2.paintOn( s, 0, 1 );
34         (new HLine( 15, 'B' )).paintOn( s, 0, 2 ); // tricky to read
35         s.draw( t );
36
37         t.println( "Clear that screen," );
38         s.clear();
39
40         t.println( "draw 3 Boxes (2 overlapping):" );
41         Box b = new Box( 6, 5, 'R' );
42         b.paintOn( s, 1, 1 );
43         b = new Box( 7, 4, 'G' ); // create a new (different) Box b
44         b.paintOn( s, 2, 3 ); // paint Box b on s
45         b.paintOn( s, 17, 5 ); // paint Box b partly off the Screen
46         s.draw( t );
47     }
48 }
```

```
1 // fo1/3/shapes/InteractiveShapes.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * Interactive program to study shapes.
8  *
9  * @version3
10 */
11
12 public class InteractiveShapes
13 {
14     public static void main( String[] args )
15     {
16         Terminal t = new Terminal();
17         Screen s = new Screen(
18             t.readInt("screen width: "),
19             t.readInt("screen height: "));
20
21         char c = 'a';
22         int x,y;
23         while ( t.readYesOrNo("more") ) {
24             char shape = t.readChar("h(1line), b(ox), c(lear): ");
25             switch (shape) {
26                 case 'h':
27                     int length = t.readInt("HLine length: ");
28                     x = t.readInt("x coordinate: ");
29                     y = t.readInt("y coordinate: ");
30                     (new HLine(length, c++)).paintOn(s,x,y);
31                     break;
32                 case 'b':
33                     int w = t.readInt("Box width: ");
34                     int h = t.readInt("Box height: ");
35                     x = t.readInt("x coordinate: ");
36                     y = t.readInt("y coordinate: ");
37                     (new Box(w,h,c++)).paintOn(s,x,y);
38                     break;
39                 case 'c':
40                     s.clear();
41                     break;
42                 default:
43                     t.println("try again");
44                     continue;
45             }
46         }
47     }
48 }
```



```

1 // fo1/3/shapes/TextLine.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5 // This file contains stubs for the methods.
6
7 /**
8  * A horizontal line of character text.
9  *
10 * @version 3
11 */
12
13 public class TextLine
14 {
15     /**
16      * Construct a TextLine.
17      *
18      * @param text the text of the line.
19      */
20     public TextLine( String text )
21     {
22     }
23
24     /**
25      * Paint this TextLine on Screen s at position (x,y).
26      *
27      * @param s the Screen on which this line is to be painted.
28      * @param x the x position for the line.
29      * @param y the y position for the line.
30      */
31     public void paintOn( Screen s, int x, int y )
32     {
33     }
34
35     /**
36      * Draw the TextLine to Screen s at position (0,0).
37      *
38      * @param s the Screen on which this line is to be painted.
39      */
40     public void paintOn( Screen s )
41     {
42         paintOn( s, 0, 0 );
43     }
44
45     /**
46      * Get the length of this line.
47      *
48      * @return the length in (character) pixels.
49      */
50     public int getLength()
51     {
52     }
53
54 }
55
56

```

```

57     }
58     return 0; // replace with the right answer
59 }
60 /**
61  * Unit test for class TextLine,
62  * assuming Screen and Terminal work.
63  */
64 public static void main( String[] args )
65 {
66 }
67 }
68 }

```

```

1 // foj/3/shapes/Screen.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A Screen is a (width*height) grid of (character) 'pixels'
8  * on which we may paint various shapes.  It can be drawn to
9  * a Terminal.
10 *
11 * @version 3
12 */
13
14 public class Screen
15 {
16     /**
17      * The character used to paint the screen's frame.
18      */
19
20     private static final char FRAMECHAR = '+';
21     private static final char BLANK = ' ';
22     private int width;
23     private int height;
24     private char[][] pixels;
25
26     /**
27      * Construct a Screen.
28      *
29      * @param width the number of pixels in the x direction.
30      * @param height the number of pixels in the y direction.
31      */
32
33     public Screen( int width, int height )
34     {
35         this.width = width;
36         this.height = height;
37         pixels = new char[width][height];
38         clear();
39     }
40
41     /**
42      * Clear the Screen, painting a blank at every pixel.
43      */
44
45     public void clear()
46     {
47         for (int x = 0; x < width; x++) {
48             for (int y = 0; y < height; y++) {
49                 pixels[x][y] = BLANK;
50             }
51         }
52     }
53
54     /**
55      * Paint a character pixel at position (x,y).

```

```

57      * @param c the character to be painted.
58      * @param x the (horizontal) x position.
59      * @param y the (vertical) y position.
60      */
61
62     public void paintAt( char c, int x, int y )
63     {
64         if ( 0 <= x && x < width &&
65             0 <= y && y < height ) {
66             pixels[x][y] = c;
67         }
68         // Otherwise off the Screen - nothing is painted.
69     }
70
71     /**
72      * How wide is this Screen?
73      *
74      * @return the width.
75      */
76
77     public int getWidth()
78     {
79         return width;
80     }
81
82     /**
83      * How high is this Screen?
84      *
85      * @return the height.
86      */
87
88     public int getHeight()
89     {
90         return height;
91     }
92
93     /**
94      * Draw this Screen on a Terminal.
95      *
96      * @param t the Terminal on which to draw this Screen.
97      */
98
99     public void draw( Terminal t )
100    {
101        for (int col = -1; col < width+1; col++) { // top edge
102            t.print(FRAMECHAR);
103        }
104        t.println();
105        for (int row = 0; row < height; row++) {
106            t.print(FRAMECHAR);
107            for (int col = 0; col < width; col++) { // left edge
108                t.print( pixels[col][row] );
109            }
110            t.println( FRAMECHAR ); // right edge
111        }
112        for (int col = -1; col < width+1; col++) { // bottom edge

```

```
113         t.print("FRAMECHAR");  
114     }  
115     t.println();  
116 }  
117 }
```

```

1 // fo1/4/bank/Bank.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 // Lines marked "///" flag places where changes will be needed.
7
8 /// import java.util.??
9
10 /**
11  * A Bank object simulates the behavior of a simple bank/ATM.
12  * It contains a Terminal object and a collection of
13  * BankAccount objects.
14
15  * Its public method visit opens this Bank for business,
16  * prompting the customer for input.
17
18  * To create a Bank and open it for business issue the command
19  * <code>java Bank</code>.
20
21  * @see BankAccount
22  * @version 4
23  */
24
25 public class Bank
26 {
27     private String bankName; // the name of this Bank
28     private Terminal atm; // for talking with the customer
29     private int balance = 0; // total cash on hand
30     private int transactionCount = 0; // number of Bank transactions done
31
32     private BankAccount[] accountList; // collection of BankAccounts
33     // omit next line when accountList is dynamic
34     private final static int NUM_ACCOUNTS = 3;
35
36     // what the banker can ask of the bank
37
38     private static final String BANKER_COMMANDS =
39     "Banker commands: " +
40     "exit, open, customer, report, help.";
41
42     // what the customer can ask of the bank
43
44     private static final String CUSTOMER_TRANSACTIONS =
45     "Customer transactions: " +
46     "deposit, withdraw, transfer, balance, quit, help.";
47
48     /**
49     * Construct a Bank with the given name and Terminal.
50     *
51     * @param bankName the name for this Bank.
52     * @param atm this Bank's Terminal.
53     */
54
55     public Bank( String bankName, Terminal atm )
56     {

```

```

57     this.atm = atm;
58     this.bankName = bankName;
59     // initialize collection:
60     accountList = new BankAccount[NUM_ACCOUNTS]; ///
61
62     /// When accountList is an array, fill it here.
63     /// When it's an ArrayList or a TreeMap, delete these lines.
64     /// Bank starts with no accounts, banker creates them with
65     /// the openNewAccount method.
66     accountList[0] = new BankAccount( 0, this);
67     accountList[1] = new BankAccount(100, this);
68     accountList[2] = new BankAccount(200, this);
69
70 }
71
72 /**
73  * Simulates interaction with a Bank.
74  * Presents the user with an interactive loop, prompting for
75  * banker transactions and in case of the banker transaction
76  * "customer", an account id and further customer
77  * transactions.
78  */
79
80 public void visit()
81 {
82     instructUser();
83
84     String command;
85     while ( !command =
86         atm.readWord("banker command: ").equals("exit")) {
87
88         if (command.startsWith("h")) {
89             help( BANKER_COMMANDS );
90         }
91         else if (command.startsWith("o")) {
92             openNewAccount();
93         }
94         else if (command.startsWith("r")) {
95             report();
96         }
97         else if (command.startsWith("c" ) ) {
98             BankAccount acct = whichAccount();
99             if ( acct != null )
100                 processTransactionsForAccount( acct );
101         }
102         else {
103             // Unrecognized Request
104             atm.println( "unknown command: " + command );
105         }
106     }
107     report();
108     atm.println( "Goodbye from " + bankName );
109 }
110
111 // Open a new bank account,
112 // prompting the user for information.

```

```

113 private void openNewAccount()
114 {
115     /// when accountList is a dynamic collection
116     /// remove the next two lines, uncomment and complete
117     /// the code between /* and */
118     atm.println(bankName + " is accepting no new customers\n");
119     return;
120 }
121 /*
122 // prompt for initial deposit
123 int startup = atm.readInt( "Initial deposit: " );
124 // create newAccount = new BankAccount( startup, this );
125 BankAccount newAccount = new BankAccount( startup, this );
126 // and add it to accountList
127 ???
128 // inform user
129 atm.println( "opened new account " + ??? // name or number
130             + " with $" + newAccount.getBalance());
131 */
132 }
133 // Prompt the customer for transaction to process.
134 // Then send an appropriate message to the account.
135 private void processTransactionsForAccount( BankAccount acct )
136 {
137     help( CUSTOMER_TRANSACTIONS );
138     String transaction;
139     while ( !(transaction =
140             atm.readWord( " transaction: ")).equals("quit")) {
141         if ( transaction.startsWith( "h" ) ) {
142             help( CUSTOMER_TRANSACTIONS );
143         }
144         else if ( transaction.startsWith( "d" ) ) {
145             int amount = atm.readInt( " amount: " );
146             atm.println( " deposited " + acct.deposit( amount ) );
147         }
148         else if ( transaction.startsWith( "w" ) ) {
149             int amount = atm.readInt( " amount: " );
150             atm.println( " withdrew " + acct.withdraw( amount ) );
151         }
152         else if ( transaction.startsWith( "t" ) ) {
153             atm.print( " to " );
154             BankAccount toacct = whichAccount();
155             if (toacct != null) {
156                 int amount = atm.readInt( " amount to transfer: " );
157                 atm.println( " transferred " +
158                             toacct.deposit(acct.withdraw(amount));
159             }
160         }
161     }
162 }
163 }
164 }
165 }
166 }
167 }
168 }

```

```

169     else if (transaction.startsWith("b")) {
170         atm.println(" current balance " +
171                     acct.requestBalance());
172     }
173     else {
174         atm.println(" sorry, unknown transaction" );
175     }
176 }
177 atm.println();
178 }
179 // Prompt for an account name (or number), look it up
180 // in the account list. If it's there, return it;
181 // otherwise report an error and return null.
182 private BankAccount whichAccount()
183 {
184     /// prompt for account name or account number
185     /// (whichever is appropriate)
186     int accountNumber = atm.readInt("account number: ");
187     /// Look up account in accountList
188     /// if it's there, return it
189     /// else the following two lines should execute
190     if ( accountNumber >= 0 && accountNumber < NUM_ACCOUNTS ) {
191         return accountList[accountNumber];
192     }
193     else {
194         atm.println("not a valid account");
195         return null;
196     }
197 }
198 // Report bank activity.
199 // For each BankAccount, print the customer id (name or number),
200 // account balance and the number of transactions.
201 // Then print Bank totals.
202 private void report()
203 {
204     atm.println( "\nSummaries of individual accounts:" );
205     atm.println( "account balance transaction count" );
206     for ( int i = 0; i < NUM_ACCOUNTS; i++ ) {
207         atm.println( i + "\t" + accountList[i].getBalance() +
208                     "\t" + accountList[i].getTransactionCount());
209     }
210     atm.println( "\nBank totals" );
211     atm.print( " open accounts: " + getNumberOfAccounts() );
212     atm.println( " cash on hand: $" + getBalance());
213     atm.println( " transactions: " + getTransactionCount());
214     atm.println();
215 }
216 // Welcome the user to the bank and instruct her on
217 }
218 }
219 }
220 }
221 }
222 }
223 }
224 }

```

```

225 // her options.
226 private void instructUser()
227 {
228     atm.println( "Welcome to " + bankName );
229     atm.println( "Open some accounts and work with them. " );
230     help( BANKER_COMMANDS );
231 }
232 // Display a help string.
233
234 private void help( String helpString )
235 {
236     atm.println( helpString );
237     atm.println();
238 }
239
240 /**
241  * Increment bank balance by given amount.
242  */
243 * @param amount the amount increment.
244 */
245 public void incrementBalance(int amount)
246 {
247     balance += amount;
248 }
249
250 /**
251  * Increment by one the count of transactions,
252  * for this bank.
253  */
254 public void countTransaction()
255 {
256     transactionCount++;
257 }
258
259 /**
260  * Get the number of transactions performed by this bank.
261  */
262 * @return number of transactions performed.
263 */
264 public int getTransactionCount()
265 {
266     return transactionCount;
267 }
268
269 /**
270  * Get the current bank balance.
271  */
272 * @return current bank balance.
273 */
274 public int getBalance()
275
276
277
278
279
280

```

```

281 {
282     return balance;
283 }
284
285 /**
286  * Get the current number of open accounts.
287  */
288 * @return number of open accounts.
289 */
290 public int getNumberOfAccounts()
291 {
292     return NUM_ACCOUNTS; // needs changing ...
293 }
294
295 /**
296  * Run the simulation by creating and then visiting a new Bank.
297  * <p>
298  * A -e argument causes the input to be echoed.
299  * This can be useful for executing the program against
300  * a test script, e.g.,
301  * <pre>
302  * java Bank -e < Bank.in
303  * </pre>
304  *
305  * @param args the command line arguments:
306  *     <pre>
307  *     -e echo input.
308  *     bankName any other command line argument.
309  *     </pre>
310  */
311
312 public static void main( String[] args )
313 {
314     // parse the command line arguments for the echo
315     // flag and the name of the bank
316     boolean echo = false; // default does not echo
317     String bankName = "River Bank"; // default bank name
318     for (int i = 0; i < args.length; i++ ) {
319         if (args[i].equals("-e") ) {
320             echo = true;
321         }
322         else {
323             bankName = args[i];
324         }
325     }
326     Bank aBank = new Bank( bankName, new Terminal(echo) );
327     aBank.visit();
328 }
329
330
331
332

```

```

1 // foj/4/bank/BankAccount.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A BankAccount object has private fields to keep track
8  * of its current balance, the number of transactions
9  * performed and the Bank in which it is an account, and
10 * and public methods to access those fields appropriately.
11 *
12 * @see Bank
13 * @version 4
14 */
15
16 public class BankAccount
17 {
18     private int balance = 0; // Account balance (whole dollars)
19     private int transactionCount = 0; // Number of transactions performed
20     private Bank issuingBank; // Bank issuing this account
21
22     /**
23      * Construct a BankAccount with the given initial balance and
24      * issuing Bank. Construction counts as this BankAccount's
25      * first transaction.
26      *
27      * @param initialBalance the opening balance.
28      * @param issuingBank the bank that issued this account.
29      */
30
31     public BankAccount( int initialBalance, Bank issuingBank )
32     {
33         this.issuingBank = issuingBank;
34         deposit( initialBalance );
35     }
36
37     /**
38      * Withdraw the given amount, decreasing this BankAccount's
39      * balance and the issuing Bank's balance.
40      * Counts as a transaction.
41      *
42      * @param amount the amount to be withdrawn
43      * @return amount withdrawn
44      */
45
46     public int withdraw( int amount )
47     {
48         incrementBalance( -amount );
49         countTransaction();
50         return amount ;
51     }
52
53     /**
54      * Deposit the given amount, increasing this BankAccount's
55      * balance and the issuing Bank's balance.
56      * Counts as a transaction.

```

```

57
58     * @param amount the amount to be deposited
59     * @return amount deposited
60     */
61
62     public int deposit( int amount )
63     {
64         incrementBalance( amount );
65         countTransaction();
66         return amount ;
67     }
68
69     /**
70      * Request for balance. Counts as a transaction.
71      *
72      * @return current account balance
73      */
74
75     public int requestBalance()
76     {
77         countTransaction();
78         return getBalance() ;
79     }
80
81     /**
82      * Get the current balance.
83      * Does NOT count as a transaction.
84      *
85      * @return current account balance
86      */
87
88     public int getBalance()
89     {
90         return balance;
91     }
92
93     /**
94      * Increment account balance by given amount.
95      * Also increment issuing Bank's balance.
96      * Does NOT count as a transaction.
97      *
98      * @param amount the amount increment.
99      */
100
101     public void incrementBalance( int amount )
102     {
103         balance += amount;
104         this.getIssuingBank().incrementBalance( amount );
105     }
106
107     /**
108      * Get the number of transactions performed by this
109      * account. Does NOT count as a transaction.
110      *
111      * @return number of transactions performed.
112     */

```

```
113 public int getTransactionCount()
114 {
115     return transactionCount;
116 }
117
118 /**
119  * Increment by 1 the count of transactions, for this account
120  * and for the issuing Bank.
121  * Does NOT count as a transaction.
122  */
123
124 public void countTransaction()
125 {
126     transactionCount++;
127     this.getIssuingBank().countTransaction();
128 }
129
130 /**
131  * Get the bank that issued this account.
132  * Does NOT count as a transaction.
133  * @return issuing bank.
134  */
135
136 public Bank getIssuingBank()
137 {
138     return issuingBank;
139 }
140
141 }
142 }
```



```
1 open
2 1000
3 open
4 2000
5 help
6 report
7 open
8 3000
9 customer
10 0
11 balance
12 deposit
13 9999
14 balance
15 quit
16 customer
17 1
18 transfer
19 9
20 transfer
21 2
22 45
23 quit
24 exit
```

```

1 Welcome to River Bank
2 Open some accounts and work with them.
3 Banker commands: exit, open, customer, report, help.
4
5 banker command: open
6 Initial deposit: 1000
7 opened new account 0 with $1000
8 banker command: open
9 Initial deposit: 2000
10 opened new account 1 with $2000
11 banker command: help
12 Banker commands: exit, open, customer, report, help.
13
14 banker command: report
15
16 Summaries of individual accounts:
17 account balance transaction count
18 0 $1000 1
19 1 $2000 1
20
21 Bank totals
22 open accounts: 2
23 cash on hand: $3000
24 transactions: 2
25
26 banker command: open
27 Initial deposit: 3000
28 opened new account 2 with $3000
29 banker command: customer
30 account number: 0
31 Customer transactions: deposit, withdraw, transfer, balance, quit, he
32
33 transaction: balance
34 current balance 1000
35 transaction: deposit
36 amount: 9999
37 deposited 9999
38 transaction: balance
39 current balance 10999
40 transaction: quit
41
42 banker command: customer
43 account number: 1
44 Customer transactions: deposit, withdraw, transfer, balance, quit, he
45
46 transaction: transfer
47 to account number: 9
48 not a valid account
49 transaction: transfer
50 to account number: 2
51 amount to transfer: 45
52 transferred 45
53 transaction: quit
54
55 banker command: exit
56

```

```

57 Summaries of individual accounts:
58 account balance transaction count
59 0 $10999 4
60 1 $1955 2
61 2 $3045 2
62
63 Bank totals
64 open accounts: 3
65 cash on hand: $15999
66 transactions: 8
67
68 Goodbye from River Bank

```

```
1 open
2 grouchu
3 1000
4 customer
5 harpo
6 open
7 harpo
8 2000
9 help
10 report
11 open
12 chico
13 3000
14 customer
15 grouchu
16 balance
17 deposit
18 9999
19 balance
20 quit
21 customer
22 harpo
23 transfer
24 chico
25 45
26 quit
27 exit
```

```

1 Welcome to River Bank
2 Open some accounts and work with them.
3 Banker commands: exit, open, customer, report, help.
4
5 banker command: open
6 Account name: groucho
7 Initial deposit: 1000
8 opened new account groucho with $1000
9 banker command: customer
10 account name: harpo
11 not a valid account
12 banker command: open
13 Account name: harpo
14 Initial deposit: 2000
15 opened new account harpo with $2000
16 banker command: help
17 Banker commands: exit, open, customer, report, help.
18
19 banker command: report
20
21 Summaries of individual accounts:
22 account balance transaction count
23 groucho $1000 1
24 harpo $2000 1
25
26 Bank totals
27 open accounts: 2
28 cash on hand: $3000
29 transactions: 2
30
31 banker command: open
32 Account name: chico
33 Initial deposit: 3000
34 opened new account chico with $3000
35 banker command: customer
36 account name: groucho
37 Customer transactions: deposit, withdraw, transfer, balance, quit, he
38
39 transaction: balance
40 current balance 1000
41 transaction: deposit
42 amount: 9999
43 deposited 9999
44 transaction: balance
45 current balance 10999
46 transaction: quit
47
48 banker command: customer
49 account name: harpo
50 Customer transactions: deposit, withdraw, transfer, balance, quit, he
51
52 transaction: transfer
53 to account name: chico
54 amount to transfer: 45
55 transferred 45
56 transaction: quit

```

```

57 banker command: exit
58
59 Summaries of individual accounts:
60 account balance transaction count
61 chico $3045 2
62 groucho $10999 4
63 harpo $1955 2
64
65 Bank totals
66 open accounts: 3
67 cash on hand: $15999
68 transactions: 8
69
70 Goodbye from River Bank
71

```

```
1 // foj/examples/Reverse.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.util.ArrayList;
7
8 /**
9  * Reverse the order of lines entered from standard input.
10  */
11
12 public class Reverse
13 {
14
15     /**
16      * Read lines typed at the terminal until end-of-file,
17      * saving them in an ArrayList.
18      *
19      * Then print the lines in reverse order.
20      */
21
22     public static void main( String[] args )
23     {
24         Terminal t = new Terminal();
25         ArrayList list = new ArrayList();
26         String line;
27
28         while ((line = t.readLine()) != null ) {
29             list.add(line);
30         }
31
32         for (int i = list.size()-1; i >= 0; i--) {
33             line = (String)list.get(i);
34             t.println( line );
35         }
36     }
37 }
```

```

1 // foj/4/dictionary/Dictionary.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * Model a dictionary with a TreeMap of (word, Definition) pairs.
10  *
11  * @see Definition
12  *
13  * @version 4
14  */
15
16 public class Dictionary
17 {
18     private TreeMap entries;
19
20     /**
21      * Construct an empty Dictionary.
22      */
23     public Dictionary()
24     {
25         entries = new TreeMap();
26     }
27
28     /**
29      * Add an entry to this Dictionary.
30      *
31      * @param word the word being defined.
32      * @param definition the Definition of that word.
33      */
34     public void addEntry( String word, Definition definition )
35     {
36         entries.put( word, definition );
37     }
38
39     /**
40      * Look up an entry in this Dictionary.
41      *
42      * @param word the word whose definition is sought
43      * @return the Definition of that word, null if none.
44      */
45     public Definition getEntry( String word )
46     {
47         return (Definition)entries.get(word);
48     }
49
50     /**
51      * Get the size of this Dictionary.
52      *
53      * @return the number of words.
54
55
56

```

```

57     */
58     public int getSize()
59     {
60         return entries.size();
61     }
62
63     /**
64      * Construct a String representation of this Dictionary.
65      *
66      * @return a multiline String representation.
67      */
68     public String toString()
69     {
70         String str = "";
71         String word;
72         Definition definition;
73         Set allWords = entries.keySet();
74         Iterator wordIterator = allWords.iterator();
75         while ( wordIterator.hasNext() ) {
76             word = (String)wordIterator.next();
77             definition = this.getEntry( word );
78             str += word + ":\n" + definition.toString() + "\n";
79         }
80         return str;
81     }
82
83 }
84

```

```
1 // fo1/4/dictionary/Definition.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * Model the definition of a word in a dictionary.
8  *
9  * @see Dictionary
10 *
11 * @version 4
12 */
13
14 public class Definition
15 {
16     private String definition; // the defining string
17
18     /**
19      * Construct a simple Definition.
20      *
21      * @param definition the definition.
22      */
23
24     public Definition( String definition )
25     {
26         this.definition = definition;
27     }
28
29     /**
30      * Construct a String representation of this Definition.
31      *
32      * @return the definition string.
33      */
34
35     public String toString()
36     {
37         return definition;
38     }
39 }
```

```

1 // fo1/4/dictionary/lookup.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * On line word lookup.
8  *
9  * @see Dictionary
10 * @see Definition
11 *
12 * @version 4
13 */
14
15 public class Lookup
16 {
17     private static Terminal t = new Terminal();
18     private static Dictionary dictionary = new Dictionary();
19
20     /**
21      * Helper method to fill the dictionary with some simple
22      * Definitions.
23      *
24      * A real Dictionary would live in a file somewhere.
25      */
26
27     private static void fillDictionary()
28     {
29         dictionary.addEntry( "shape",
30             new Definition( "a geometric object in a plane" ) );
31         dictionary.addEntry( "quadrilateral",
32             new Definition( "a polygonal shape with four sides" ) );
33         dictionary.addEntry( "rectangle",
34             new Definition( "a right-angled quadrilateral" ) );
35         dictionary.addEntry( "square",
36             new Definition( "a rectangle having equal sides" ) );
37     }
38
39     /**
40      * Helper method to print the Definition of a single word,
41      * or a message if the word is not in the Dictionary.
42      *
43      * @param word the word whose definition is wanted.
44      */
45
46     private static void printDefinition(String word)
47     {
48         Definition definition = dictionary.getEntry(word);
49         if (definition == null) {
50             t.println("sorry, no definition found for " + word);
51         }
52         else {
53             t.println(definition.toString());
54         }
55     }
56

```

```

57     /**
58      * Run the Dictionary lookup.
59      *
60      * Parse command line arguments for words to look up,
61      * "all" prints the whole Dictionary.
62      *
63      * Then prompt for more words, "quit" to finish.
64      *
65      * For example,
66      * <pre>
67      *
68      * %> java Lookup shape square circle
69      * shape:
70      * a geometric object in a plane
71      * square:
72      * a rectangle having equal sides
73      * circle:
74      * sorry, no definition found for circle
75      *
76      * look up words, "quit" to quit
77      * word> rectangle
78      * a right-angled quadrilateral
79      * word> quit
80      * %>
81      * </pre>
82
83      * @param args the words that we want looked up, supplied as
84      * command line arguments. If the word "all" is
85      * included, all words are looked up.
86      */
87
88     public static void main( String[] args )
89     {
90         // fill the dictionary (not a big one!)
91         fillDictionary();
92
93         // look up some words
94         String word;
95
96         // words specified on command line
97         for (int i = 0; i < args.length; i++) {
98             word = args[i];
99             if (word.equals("all")) {
100                 t.println("The whole dictionary ( " +
101                     dictionary.getSize() + " entries):");
102                 t.println("-----");
103                 t.println(dictionary.toString());
104                 t.println("-----");
105             }
106             else {
107                 t.println(word + ":");
108                 printDefinition(word);
109             }
110         }
111         // words entered interactively
112

```



```
113     t.println("\nlook up words, \"quit\" to quit");
114     while (true) {
115         word = t.readWord("word> ");
116         if (word.equals("quit")) {
117             break;
118         }
119         printDefinition(word);
120     }
121 }
122 }
```

```

1 // jol/3/textfiles/TextFile.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.util.Date;
7
8 /**
9  * A TextFile mimics the sort of text file that one finds
10 * on a computer's file system. It has an owner,
11 * a create date (when the file was created),
12 * a modification date (when the file was last modified),
13 * and String contents.
14
15 * @version 3
16 */
17
18 public class TextFile
19 {
20     // Private Implementation
21
22     private String owner; // Who owns the file.
23     private Date   createdAt; // When the file was created.
24     private Date   modDate; // When the file was last modified.
25     private String contents; // The text stored in the file.
26
27     // Public Interface
28
29     /**
30      * Construct a new TextFile with given owner and
31      * contents; set the creation and modification dates.
32      *
33      * @param owner the user who owns the file.
34      * @param contents the file's initial contents.
35      */
36
37     public TextFile( String owner, String contents )
38     {
39         this.owner = owner;
40         this.contents = contents;
41         createdAt = new Date(); // date and time now
42         modDate = createdAt;
43     }
44
45     /**
46      * Replace the contents of the file.
47      *
48      * @param contents the new contents.
49      */
50
51     public void setContents( String contents )
52     {
53         this.contents = contents;
54         modDate = new Date();
55     }
56

```

```

57     /**
58      * The contents of a file.
59      *
60      * @return String contents of the file.
61      */
62
63     public String getContents()
64     {
65         return contents;
66     }
67
68     /**
69      * Append text to the end of the file.
70      *
71      * @param text the text to be appended.
72      */
73
74     public void append( String text )
75     {
76         this.setContents( contents + text );
77     }
78
79     /**
80      * Append a new line of text to the end of the file.
81      *
82      * @param text the text to be appended.
83      */
84
85     public void appendLine( String text )
86     {
87         this.setContents( contents + '\n' + text );
88     }
89
90     /**
91      * The size of a file.
92      *
93      * @return the integer size of the file
94      * (the number of characters in its String contents)
95      */
96
97     public int getSize()
98     {
99         int charCount;
100         charCount = contents.length();
101         return charCount;
102     }
103
104     /**
105      * The data and time of the file's creation.
106      *
107      * @return the file's creation date and time.
108      */
109
110     public String getCreateDate()
111     {
112         return createdAt.toString();
113     }

```

```

113     }
114
115     /**
116     * The date and time of the file's last modification.
117     *
118     * @return the date and time of the file's last modification.
119     */
120
121     public String getModDate()
122     {
123         return modDate.toString();
124     }
125
126     /**
127     * The file's owner.
128     *
129     * @return the owner of the file.
130     */
131
132     public String getOwner()
133     {
134         return owner;
135     }
136
137     /**
138     * A definition of main(), used only for testing this class.
139     *
140     * Executing
141     * <pre>
142     * %> java TextFile
143     * </pre>
144     * produces the output:
145     * <pre>
146     * TextFile myTextFile contains 13 characters.
147     * Created by Bill, Sat Dec 29 14:02:37 EST 2001
148     * Hello, world.
149     *
150     * append new line "How are you today?"
151     * Hello, world.
152     * How are you today?
153     * TextFile myTextFile contains 32 characters.
154     * Modified Sat Dec 29 14:02:38 EST 2001
155     * </pre>
156     */
157
158     public static void main( String[] args )
159     {
160         Terminal terminal = new Terminal();
161         TextFile myTextFile = new TextFile( "bill", "Hello, world." );
162
163         terminal.println( "TextFile myTextFile contains " +
164                         myTextFile.getSize() + " characters." );
165         terminal.println( "Created by " + myTextFile.getOwner() +
166                         " " +
167                         myTextFile.getCreatedDate() );
168         terminal.println( myTextFile.getContents() );

```

```

169         terminal.println();
170
171         terminal.println( "append new line \"How are you today?\"" );
172         myTextFile.appendLine( "How are you today?" );
173         terminal.println( myTextFile.getContents() );
174         terminal.println( "TextFile myTextFile contains " +
175                         myTextFile.getSize() + " characters." );
176         terminal.println( "Modified " +
177                         myTextFile.getModDate() );
178     }
179 }

```

```

1 // fo1/4/textfiles/Directory.java
2 //
3 //
4 // Copyright 2003 Ehan Bolker and Bill Campbell
5 //
6 // This draft contains just stubs for the methods.
7 // You can invoke them all, but none will do anything.
8 /**
9  * Directory of TextFiles.
10  *
11  * @version 4
12  */
13
14 public class Directory
15 {
16     /**
17      * Construct a Directory.
18      */
19
20     public Directory( )
21     {
22     }
23
24     /**
25      * The size of a directory is the number of TextFiles it contains.
26      *
27      * @return the number of TextFiles.
28      */
29
30     public int getSize( )
31     {
32         return 0;
33     }
34
35     /**
36      * Add a TextFile to this Directory. Overwrite if a TextFile
37      * of that name already exists.
38      *
39      * @param name the name under which this TextFile is added.
40      * @param afile the TextFile to add.
41      */
42
43     public void addTextFile(String name, TextFile afile)
44     {
45     }
46
47     /**
48      * Get a TextFile in this Directory, by name .
49      *
50      * @param filename the name of the TextFile to find.
51      * @return the TextFile found, null if none.
52      */
53
54     public TextFile retrieveTextFile( String filename )
55     {
56

```

```

57         return null;
58     }
59
60     /**
61      * Get the contents of this Directory as an array of
62      * the file names, each of which is a String.
63      *
64      * @return the array of names.
65      */
66
67     public String[] getFileNames( )
68     {
69         // pseudocode for an implementation:
70         // declare an array of String
71         // create that array with as many spaces as there
72         // are TextFile's in this Directory
73         // loop through the keys of the TreeMap of TextFiles,
74         // adding each String key to the array
75         // return the array
76
77         // the next line is there because we have to return
78         // something_ in order to satisfy the compiler
79         return new String[0];
80     }
81
82     /**
83      * main, for unit testing.
84      *
85      * The command
86      * <pre>
87      * java Directory
88      * </pre>
89      * should produce output
90      * <pre>
91      * bill      17      Sun Jan 06 19:40:13 EST 2003      diary
92      * eb        12      Sun Jan 06 19:40:13 EST 2003      greeting
93      * </pre>
94      * (with current dates, of course).
95      */
96
97     public static void main( String[] args )
98     {
99         Directory dir = new Directory();
100        dir.addTextFile("greeting", new TextFile("eb", "Hello, world"));
101        dir.addTextFile("diary", new TextFile("bill", "Writing Directory")
102        // now list TextFiles in dir to get output specified
103        }
104    }

```

```

1 // foj/4/estore/ESTore.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * An EStore object simulates the behavior of a simple on line
8  * shopping web site.
9
10 * It contains a Terminal object to model the customer's browser
11 * and a Catalog of Items that may be purchased and
12 * then added to the customer's shoppingCart.
13
14 * @version 4
15 */
16
17 public class EStore
18 {
19     private String  storeName;
20     private Terminal browser;
21     private Catalog catalog;
22
23     /**
24      * Construct a new EStore.
25      *
26      * @param storeName the name of the EStore
27      * @param browser the visitor's Terminal.
28      */
29
30     public EStore( String storeName, Terminal browser )
31     {
32         this.browser = browser;
33         this.storeName = storeName;
34         this.catalog = new Catalog();
35         catalog.addItem( new Item("quaffle", 55) );
36         catalog.addItem( new Item("bludger", 15) );
37         catalog.addItem( new Item("snitch", 1000) );
38     }
39
40     /**
41      * Visit this EStore.
42      *
43      * Execution starts here when the store opens for
44      * business. User can visit as a customer, act as
45      * the manager, or exit.
46      */
47
48     public void visit()
49     {
50         // Print a friendly welcome message.
51         browser.println( "Welcome to " + storeName );
52         while (true) { // an infinite loop ...
53             browser.println();
54             String whoAreYou = browser.readWord(
55                 storeName + " (manager, visit, exit): ");
56             if (whoAreYou.equals("exit")) {

```

```

57         break; // leave the while loop
58     }
59     if (whoAreYou.equals("manager")) {
60         managerVisit();
61     }
62     if (whoAreYou.equals("visit")) {
63         customerVisit();
64     }
65 }
66
67 /**
68  * Manager options:
69  *
70  * examine the catalog
71  * add an Item to the catalog
72  * quit
73 */
74 private void managerVisit( )
75 {
76     while (true) {
77         String cmd =
78             browser.readWord("manager command (show, new, quit):");
79         if (cmd.equals("quit")) {
80             break; // leave manager command while loop
81         }
82         else if (cmd.equals("show")) {
83             catalog.show(browser);
84         }
85         else if (cmd.equals("new")) {
86             String itemName = browser.readWord(" item name: ");
87             int cost = browser.readInt(" cost: ");
88             catalog.addItem( new Item(itemName, cost) );
89         }
90         else {
91             browser.println("unknown manager command: " + cmd);
92         }
93     }
94 }
95
96 /**
97  * Customer visits this EStore.
98  *
99  * Loop allowing customer to select items to add to her
100  * shoppingCart.
101  */
102
103 private void customerVisit( )
104 {
105     // Create a new, empty ShoppingCart.
106     ShoppingCart basket = new ShoppingCart();
107     browser.println( "Currently available:");
108     catalog.show(browser);
109     while ( true ) { // loop forever ...
110         String nextPurchase = browser.readWord(
111

```

```
113         "select your purchase, checkout, help: ");
114     }
115     if ( nextPurchase.equals("checkout" )) break; // leave loop!
116     if ( nextPurchase.equals("help" )) {
117         catalog.show(browser);
118         continue; // go back to top of while loop
119     }
120     // customer has entered the name of an Item
121     basket.addItem( catalog.getItem(nextPurchase) );
122 }
123
124
125     int numberPurchased = basket.getCount();
126     browser.println("We are shipping these " +
127         basket.getCount() + " Items:");
128     basket.showContents(browser);
129     browser.println("and charging your account $" + basket.getCost())
130     browser.println("Thank you for shopping at " + storeName);
131 }
132
133 /**
134  * The EStore simulation program begins here when the user
135  * issues the command <code>java EStore</code>
136  *
137  * If first command line argument is "-e" instantiate a
138  * Terminal that echoes its input.
139  *
140  * The next command line argument (if there is one)
141  * is the name of the EStore.
142  *
143  * @param args <-e> <storeName>
144  */
145     public static void main( String[] args )
146     {
147
148         String storeName = "Virtual Minimal Minimal"; //default
149
150         // check to see if first argument is "-e"
151         boolean echo = ( args.length > 0 ) && ( args[0].equals("-e") );
152
153         // if first argument was "-e" then look at second for store name
154         int nextArg = (echo ? 1 : 0 );
155
156         if (args.length > nextArg) {
157             storeName = args[nextArg];
158         }
159
160         // Print this to simulate internet search.
161         System.out.println("connecting ...");
162
163         // Create an EStore object and visit it
164         (new EStore(storeName, new Terminal(echo))).visit();
165     }
166 }
```

```

1 // foj/4/estore/ShoppingCart.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A ShoppingCart keeps track of a customer's purchases.
8  *
9  * @see EStore
10 * @version 4
11 */
12
13 public class ShoppingCart
14 {
15     /** replace these two fields by a single ArrayList
16     private int count; // number of Items in this ShoppingCart
17     private int cost; // total cost of Items in this ShoppingCart
18
19     /**
20     * Construct a new empty ShoppingCart.
21     */
22
23     public ShoppingCart()
24     {
25         count = 0;
26         cost = 0;
27     }
28
29     /**
30     * Add an Item to this ShoppingCart.
31     *
32     * @param item the Item to add.
33     */
34
35     public void addItem( Item item )
36     {
37         /** this code just keeps track of the totals
38         /** replace it with code that adds the item to the list
39         count++;
40         this.cost += item.getCost(); // Java idiom: a += b means a = a +
41         }
42
43     /**
44     * Return an Item from this ShoppingCart.
45     *
46     * @param item the Item to return.
47     */
48
49     public void returnItem( Item item )
50     {
51         /** look through the list looking for Item
52         /** remove it if it's there
53         }
54
55     /**
56     * What happens when this ShoppingCart is asked how many

```

```

57
58     * Items it contains.
59     * @return the number of items in this ShoppingCart.
60     */
61
62     public int getCount()
63     {
64         /** get this information from the list,
65         /** since the count field no longer exists
66         return count;
67     }
68
69     /**
70     * What happens when this ShoppingCart is asked the total
71     * cost of the Items it contains.
72     *
73     * @return the total cost of the items in this ShoppingCart.
74     */
75     public int getCost()
76     {
77         /** get this information from the list,
78         /** since the cost field no longer exists
79         return cost;
80     }
81
82     /**
83     * Write the contents of this ShoppingCart to a Terminal.
84     *
85     * @param t the Terminal to use for output.
86     */
87
88     public void showContents( Terminal t )
89     {
90         /** work to do here ...
91         t.println(" [sorry, can't yet print ShoppingCart contents]");
92     }
93 }

```

```
1 // fo1/4/estore/Item.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * An Item models an object that might be stocked in a store.
8  * Each Item has a cost.
9  *
10 * @version 4
11 */
12
13 public class Item
14 {
15     private int cost;
16     private String name;
17
18     /**
19      * Construct an Item object.
20      *
21      * @param name the nme of this Item.
22      * @param cost the cost of this Item.
23      */
24
25     public Item( String name, int cost )
26     {
27         this.name = name;
28         this.cost = cost;
29     }
30
31     /**
32      * How much does this Item cost?
33      *
34      * @return the cost.
35      */
36
37     public int getCost()
38     {
39         return cost;
40     }
41
42     /**
43      * What is this Item called?
44      *
45      * @return the name.
46      */
47
48     public String getName()
49     {
50         return name;
51     }
52 }
```



```

1 // foj/4/estore/Catalog.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.util.TreeMap;
7
8 /**
9  * A Catalog models the collection of Items that an
10 * EStore might carry.
11 *
12 * @see EStore
13 *
14 * @version 4
15 */
16
17 public class Catalog
18 {
19     private TreeMap items;
20
21     /**
22      * Construct a Catalog object.
23      */
24
25     public Catalog( )
26     {
27         items = new TreeMap();
28     }
29
30     /**
31      * Add an Item to this Catalog.
32      *
33      * @param item the Item to add.
34      */
35
36     public void addItem( Item item )
37     {
38         items.put( item.getName(), item );
39     }
40
41     /**
42      * Get an Item from this Catalog.
43      *
44      * @param itemName the name of the wanted Item
45      *
46      * @return the Item, null if none.
47      */
48
49     public Item getItem( String itemName )
50     {
51         return (Item)items.get(itemName);
52     }
53
54     /**
55      * Display the contents of this Catalog.
56

```

```

57      * @param t the Terminal to print to.
58      */
59
60     public void show( Terminal t )
61     {
62         // loop on items, printing name and cost
63         t.println(" [sorry, can't yet print Catalog contents]");
64     }
65 }

```

```

1 // fo1/5/shapes/Line.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A Line has a length and a paintChar used to paint
8  * itself on a Screen.
9  *
10 * Subclasses of this abstract class specify the direction
11 * of the line.
12 *
13 * @version 5
14 */
15
16 public abstract class Line
17 {
18     protected int length; // length in (character) pixels.
19     protected char paintChar; // character used for painting.
20
21     /**
22      * Construct a Line.
23      *
24      * @param length length in (character) pixels.
25      * @param paintChar character used for painting this Line.
26      */
27     protected Line( int length, char paintChar )
28     {
29         this.length = length;
30         this.paintChar = paintChar;
31     }
32
33     /**
34      * Get the length of this line.
35      *
36      * @return the length in (character) pixels.
37      */
38     public int getLength()
39     {
40         return length;
41     }
42
43     /**
44      * Set the length of this line.
45      *
46      * @param length the new length in (character) pixels.
47      */
48     public void setLength( int length )
49     {
50         this.length = length;
51     }
52
53     /**
54      */
55
56

```

```

57     * Get the paintChar of this Line.
58     *
59     * @return the paintChar.
60     */
61
62     public char getPaintChar()
63     {
64         return paintChar;
65     }
66
67     /**
68      * Set the paintChar of this Line.
69      *
70      * @param paintChar the new paintChar.
71      */
72
73     public void setPaintChar( char paintChar )
74     {
75         this.paintChar = paintChar;
76     }
77
78     /**
79      * Paint this Line on Screen s at position (x,y).
80      *
81      * @param s the Screen on which this Line is to be painted.
82      * @param x the x position for the line.
83      * @param y the y position for the line.
84      */
85
86     public abstract void paintOn( Screen s, int x, int y );
87
88     /**
89      * Paint this Line on Screen s at position (0,0).
90      *
91      * @param s the Screen on which this Line is to be painted.
92      */
93
94     public void paintOn( Screen s )
95     {
96         paintOn( s, 0, 0 );
97     }
98

```

```

1 // fo1/5/shapes/HLine.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * An HLine is a horizontal line.
8  */
9
10 public class HLine extends Line
11 {
12     /**
13      * Construct an HLine having a paintChar and a length.
14      *
15      * @param length length in (character) pixels.
16      * @param paintChar character used for painting this line.
17      */
18
19     public HLine( int length, char paintChar )
20     {
21         super( length, paintChar );
22     }
23
24     /**
25      * Paint this Line on Screen s at position (x,y).
26      *
27      * @param screen the Screen on which this Line is to be painted.
28      * @param x       the x position for the line.
29      * @param y       the y position for the line.
30      */
31
32     public void paintOn( Screen screen, int x, int y )
33     {
34         for ( int i = 0; i < length; i++ )
35             screen.paintAt( paintChar, x+i, y );
36     }
37
38     /**
39      * Unit test for class HLine.
40      */
41
42     public static void main( String[] args )
43     {
44         Terminal terminal = new Terminal();
45
46         terminal.println( "Self documenting unit test of HLine." );
47         terminal.println( "The two Screens that follow should match." );
48         terminal.println();
49         terminal.println( "Hard coded picture:" );
50         terminal.println( "+++++++" );
51         terminal.println( "+++++++" );
52         terminal.println( "+++++++" );
53         terminal.println( "+++++++" );
54         terminal.println( "+++++++" );
55         terminal.println( "+++++++" );
56         terminal.println( "+++++++" );

```

```

57         terminal.println( "+" );
58         terminal.println( "+++++++" );
59         terminal.println();
60
61         terminal.println( "Picture drawn using HLine methods:" );
62         Screen screen = new Screen( 20, 6 );
63
64         Line hline = new HLine( 10, 'x' );
65         hline.paintOn( screen );
66
67         hline.setLength( 5 );
68         hline.paintOn( screen, 0, 1 );
69
70         hline.setPaintChar( '*' );
71         hline.paintOn( screen, 3, 3 );
72
73         hline.setLength( 1 );
74         hline.setPaintChar( '1' );
75         hline.paintOn( screen, 4, 4 );
76
77         screen.draw( terminal );
78
79     }
80 }

```

```

1 // jol/5/shapes/VLine.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A VLine is a vertical Line.
8  */
9
10 public class VLine extends Line
11 {
12     /**
13      * Construct a VLine having a paintChar and a length.
14      *
15      * @param length length in (character) pixels.
16      * @param paintChar character used for painting this line.
17      */
18
19     public VLine( int length, char paintChar )
20     {
21         super( length, paintChar );
22     }
23
24     /**
25      * Paint this Line on Screen s at position (x,y).
26      *
27      * @param screen the Screen on which this Line is to be painted.
28      * @param x       the x position for the line.
29      * @param y       the y position for the line.
30      */
31
32     public void paintOn( Screen screen, int x, int y )
33     {
34         for ( int i = 0; i < length; i++ )
35             screen.paintAt( paintChar, x, y+i );
36     }
37
38     /**
39      * Unit test for class VLine.
40      */
41
42     public static void main( String[] argv )
43     {
44         Terminal terminal = new Terminal();
45
46         terminal.println( "Self documenting unit test of VLine." );
47         terminal.println( "The two Screens that follow should match." );
48         terminal.println();
49         terminal.println( "Hard coded picture:" );
50         terminal.println( "+++++++" );
51         terminal.println( "+xx  +");
52         terminal.println( "+xx  +");
53         terminal.println( "+xx  +");
54         terminal.println( "+xx  +");
55         terminal.println( "+xx *1 +");
56         terminal.println( "+x  * +");

```

```

57         terminal.println( "+x  * +");
58         terminal.println( "+  * +");
59         terminal.println( "+  +");
60         terminal.println( "+++++++" );
61         terminal.println();
62
63         terminal.println( "Picture drawn using VLine methods:" );
64         Screen screen = new Screen( 7, 9 );
65
66         Line vLine = new VLine( 7, 'x' );
67         vLine.paintOn( screen );
68
69         vLine.setLength(5);
70         vLine.paintOn( screen, 1, 0 );
71
72         vLine.setPaintChar( '*' );
73         vLine.paintOn( screen, 3, 3 );
74
75         vLine.setLength(1);
76         vLine.setPaintChar( '1' );
77         vLine.paintOn( screen, 4, 4 );
78
79         screen.draw( terminal );
80
81     }
82 }

```

```

1 // fo1/5/shapes/ShapeOnScreen.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 // This file is used in one of the Chapter 5 exercises on shapes.
7
8 /**
9  * A ShapeOnScreen models a Shape to be painted at
10 * a given position on a Screen.
11 *
12 * @see Shape
13 * @see Screen
14 * @version 5
15 */
16
17
18 public class ShapeOnScreen
19 {
20     private Shape shape;
21     private int x;
22     private int y;
23
24     /**
25      * Construct a ShapeOnScreen.
26      *
27      * @param shape the Shape
28      * @param x its x coordinate
29      * @param y its y coordinate
30      */
31
32     public ShapeOnScreen( Shape shape, int x, int y )
33     {
34         this.shape = shape;
35         this.x     = x;
36         this.y     = y;
37     }
38
39     /**
40      * What Shape does this ShapeOnScreen represent?
41      *
42      * @return the Shape.
43      */
44
45     public Shape getShape() {
46         return shape;
47     }
48
49     /**
50      * The current x coordinate of this ShapeOnScreen.
51      *
52      * @return the x coordinate.
53      */
54
55     public int getX() {
56         return x;

```

```

57     }
58
59     /**
60      * The current y coordinate of this ShapeOnScreen.
61      *
62      * @return the y coordinate.
63      */
64
65     public int getY() {
66         return y;
67     }
68
69     /**
70      * Unit test.
71      */
72
73     public static void main( String[] args ) {
74         ShapeOnScreen sos = new ShapeOnScreen( null, 5, 7);
75         System.out.println("Shape: " + sos.getShape());
76         System.out.println("x: " + sos.getX());
77         System.out.println("y: " + sos.getY());
78     }
79 }

```

```

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

```

```

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

```

```

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

```

```

169
170 /**
171  * A JFile whose parent is null is defined to be the root
172  * (of a tree).
173  */
174 * @return true when this JFile is the root.
175 */
176
177 public boolean isRoot()
178 {
179     return (parent == null);
180 }
181
182 /**
183  * How a JFile represents itself as a String.
184  * That is,
185  * <pre>
186  *   owner      size      modDate      name+suffix
187  * </pre>
188  */
189 * @return the String representation.
190 */
191
192 public String toString()
193 {
194     return getOwner() + "\t" +
195            getSize() + "\t" +
196            getModDate() + "\t" +
197            getName() + getSuffix();
198 }
199
200 // Unit test: main() and static support
201
202 private static Terminal terminal = new Terminal();
203
204 /**
205  * A unit test of JFile and its subclasses.
206  */
207
208 public static void main( String[] args )
209 {
210     out("Some hardwired, self documenting JFile system tests");
211     out("create and then explore JFile hierarchy");
212     out("    root      (owner sysadmin)");
213     out("    billhome (owner bill)");
214     out("    ebhome   (owner eb)");
215     out("    cs110    (owner eb)");
216     out("    diary   (owner eb)");
217     out("    insult  (owner bill)");
218     Directory root = new Directory( " ", "sysadmin", null );
219     Directory home1 = new Directory( "ebhome", "eb", root );
220     Directory home2 = new Directory( "billhome", "bill", root );
221
222     TextFile insult = new TextFile( "insult", "bill", home1,
223                                     "Your mother wore sneakers," );
224     insult.append( "\nIn the shower." );

```

```

225
226 Directory cs110 = new Directory( "cs110", "eb", home1);
227 cs110.addJFile( "diary",
228               new TextFile( "diary", "eb", cs110,
229                           "started work on Chapter 3"));
230
231 out("\nlist contents of the root directory:");
232 list( root );
233
234 out("\nlist contents of ebhome:");
235 list( home1 );
236
237 out("\nretrieve billhome, list its contents (empty):");
238 list( (Directory) root.retrieveJFile("billhome") );
239
240 out("\nretrieve insult, contents two line insult:");
241 type( (TextFile)home1.retrieveJFile("insult"));
242
243 out("\nretrieve file \"foo\" from ebhome, try to display it:");
244 type( (TextFile)home1.retrieveJFile("foo") );
245
246 out("\nlist contents of cs110 (one file):");
247 list( (Directory) home1.retrieveJFile("cs110") );
248
249 out("path to root:\t " + root.getPathName() );
250 out("path to ebhome:\t " + home1.getPathName() );
251 out("path to cs110:\t " + cs110.getPathName() );
252
253
254 // display a listing of the contents of a Directory
255
256 private static void list( Directory dir )
257 {
258     terminal.println( dir.getName() );
259     terminal.println( dir.getSize() +
260                    (dir.getSize() == 1
261                     ? " file:" : " files:") );
262
263     String[] fileNames = dir.getFileNames();
264     for ( int i = 0; i < fileNames.length; i++ ) {
265         String fileName = fileNames[i];
266         JFile jfile = dir.retrieveJFile( fileName );
267         terminal.println( jfile.toString() );
268     }
269
270 }
271
272 // display the contents of a TextFile
273
274 private static void type( TextFile file )
275 {
276     String whatToPrint;
277     if (file == null) {
278         whatToPrint = "no such file";
279     }
280     else {
281         whatToPrint = file.getContents();

```

```

281     }
282     terminal.println( whatToPrint );
283 }
284 // abbreviation for "terminal.println"
285
286 private static void out( String s )
287 {
288     terminal.println( s );
289 }
290
291 }

```



```

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

```

```

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

```

```

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

```

```

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

```

```

1 // fo1/5/bank/Bank.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * A Bank object simulates the behavior of a simple bank/ATM.
10 * It contains a Terminal object and a collection of
11 * BankAccount objects.
12 *
13 * The visit method opens this Bank for business,
14 * prompting the customer for input.
15 *
16 * To create a Bank and open it for business issue the command
17 * <code>java Bank</code>.
18 *
19 * @see BankAccount
20 * @version 5
21 */
22
23 public class Bank
24 {
25     private String bankName; // the name of this Bank
26     private Terminal atm; // for talking with the customer
27     private int balance = 0; // total cash on hand
28     private int transactionCount = 0; // number of Bank transactions
29     private Month month; // the current month.
30
31     private TreeMap accountList; // mapping names to accounts.
32
33     // what the banker can ask of the bank
34
35     private static final String BANKER_COMMANDS =
36     "Banker commands: " +
37     "exit, open, customer, report, help.";
38
39     // what the customer can ask of the bank
40
41     private static final String CUSTOMER_TRANSACTIONS =
42     " Customer transactions: " +
43     "deposit, withdraw, transfer, balance, cash check, quit, help.";
44
45     /**
46     * Construct a Bank with the given name and Terminal.
47     *
48     * @param bankName the name for this Bank.
49     * @param atm this Bank's Terminal.
50     */
51
52     public Bank( String bankName, Terminal atm )
53     {
54         this.atm = atm;
55         this.bankName = bankName;
56         accountList = new TreeMap();

```

```

57         month = new Month();
58     }
59
60     /**
61     * Simulates interaction with a Bank.
62     * Presents the user with an interactive loop, prompting for
63     * banker transactions and in case of the banker transaction
64     * "customer", an account id and further customer
65     * transactions.
66     */
67
68     public void visit()
69     {
70         instructUser();
71
72         String command;
73         while ( !command =
74             atm.readWord("banker command: ").equals("exit") ) {
75
76             if (command.startsWith("h") ) {
77                 help( BANKER_COMMANDS );
78             }
79             else if (command.startsWith("o") ) {
80                 openNewAccount();
81             }
82             else if (command.startsWith("r") ) {
83                 report();
84             }
85             else if (command.startsWith("c" ) ) {
86                 BankAccount acct = whichAccount();
87                 if ( acct != null )
88                     processTransactionsForAccount( acct );
89             }
90             else {
91                 // Unrecognized Request
92                 atm.println( "unknown command: " + command );
93             }
94         }
95         report();
96         atm.println( "Goodbye from " + bankName );
97     }
98
99     // Open a new bank account,
100     // prompting the user for information.
101     private void openNewAccount()
102     {
103         String accountName = atm.readWord( "Account name: " );
104         char accountType =
105             atm.readChar( "Checking/Fee/Regular? (c/F/r): " );
106         int startup = atm.readInt( "Initial deposit: " );
107         BankAccount newAccount;
108         switch( accountType ) {
109             case 'c':
110                 newAccount = new CheckingAccount( startup, this );
111             case 'f':
112

```

```

113     break;
114     case 'f':
115         newAccount = new FeeAccount( startup, this );
116         break;
117     case 'r':
118         newAccount = new RegularAccount( startup, this );
119         break;
120     default:
121         atm.println("invalid account type: " + accountType);
122         return;
123     }
124     accountList.put( accountName, newAccount );
125     atm.println( "opened new account " + accountName
126                 + " with $" + startup );
127 }
128
129 // Prompt the customer for transaction to process.
130 // Then send an appropriate message to the account.
131
132 private void processTransactionsForAccount( BankAccount acct )
133 {
134     help( CUSTOMER_TRANSACTIONS );
135     String transaction;
136     while ( !(transaction =
137            atm.readWord(" transaction: ")).equals("quit")) {
138
139         if ( transaction.startsWith( "h" ) ) {
140             help( CUSTOMER_TRANSACTIONS );
141         }
142         else if ( transaction.startsWith( "d" ) ) {
143             int amount = atm.readInt( " amount: " );
144             atm.println( " deposited " + acct.deposit( amount );
145         }
146         else if ( transaction.startsWith( "w" ) ) {
147             int amount = atm.readInt( " amount: " );
148             atm.println( " withdrew " + acct.withdraw( amount );
149         }
150         else if ( transaction.startsWith( "c" ) ) {
151             int amount = atm.readInt( " amount of check: " );
152             atm.println( " cashed check for " +
153                ((CheckingAccount)acct).honorCheck( amount ) )
154         }
155         else if ( transaction.startsWith( "t" ) ) {
156             atm.print( " to " );
157             BankAccount toacct = whichAccount();
158             if ( toacct != null ) {
159                 int amount = atm.readInt( " amount to transfer: " );
160                 atm.println( " transferred " +
161                    toacct.deposit( acct.withdraw( amount ) ) );
162             }
163         }
164         else if ( transaction.startsWith( "b" ) ) {
165             atm.println( " current balance " +
166                acct.requestBalance());
167         }
168     }

```

```

169     else {
170         atm.println(" sorry, unknown transaction" );
171     }
172 }
173 atm.println();
174 }
175
176 // Prompt for an account name (or number), look it up
177 // in the account list. If it's there, return it;
178 // otherwise report an error and return null.
179
180 private BankAccount whichAccount()
181 {
182     String accountName = atm.readWord( "account name: " );
183     BankAccount account = (BankAccount) accountList.get( accountName );
184     if ( account == null ) {
185         atm.println("not a valid account");
186     }
187     return account;
188 }
189
190 // Action to take when a new month starts.
191 // Update the month field by sending a next message.
192 // Loop on all accounts, sending each a newMonth message.
193
194 private void newMonth()
195 {
196     month.next();
197     // for each account
198     // account.newMonth()
199 }
200
201 // Report bank activity.
202 // For each BankAccount, print the customer id (name or number),
203 // account balance and the number of transactions.
204 // Then print Bank totals.
205
206 private void report()
207 {
208     atm.println( "BankName + " report for " + month );
209     atm.println( "\nSummaries of individual accounts:" );
210     atm.println( "account balance transaction count" );
211     for ( Iterator i = accountList.keySet().iterator();
212           i.hasNext(); ) {
213         String accountName = (String) i.next();
214         BankAccount acct = (BankAccount) accountList.get( accountName )
215         atm.println( accountName + "\t$" + acct.getBalance() + "\t\t"
216            acct.getTransactionCount());
217     }
218     atm.println( "\nBank totals" );
219     atm.println( " open accounts: " + getNumberOfAccounts() );
220     atm.println( " cash on hand: $" + getBalance());
221     atm.println( " transactions: " + getTransactionCount());
222     atm.println();
223 }
224

```

```

225
226 // Welcome the user to the bank and instruct her on
227 // her options.
228
229 private void instructUser()
230 {
231     atm.println( "Welcome to " + bankName );
232     atm.println( "Open some accounts and work with them." );
233     help( BANKER_COMMANDS );
234 }
235
236 // Display a help string.
237
238 private void help( String helpString )
239 {
240     atm.println( helpString );
241     atm.println();
242 }
243
244 /**
245  * Increment bank balance by given amount.
246  * @param amount the amount increment.
247  */
248
249 public void incrementBalance(int amount)
250 {
251     balance += amount;
252 }
253
254 /**
255  * Increment by one the count of transactions,
256  * for this bank.
257  */
258
259 public void countTransaction()
260 {
261     transactionCount++;
262 }
263
264 /**
265  * Get the number of transactions performed by this bank.
266  */
267
268 * @return number of transactions performed.
269 */
270
271 public int getTransactionCount()
272 {
273     return transactionCount ;
274 }
275
276 /**
277  * Get the current bank balance.
278  * @return current bank balance.
279  */
280

```

```

281
282 public int getBalance()
283 {
284     return balance;
285 }
286
287 /**
288  * Get the current number of open accounts.
289  * @return number of open accounts.
290  */
291
292 public int getNumberOfAccounts()
293 {
294     return accountList.size();
295 }
296
297 /**
298  * Run the simulation by creating and then visiting a new Bank.
299  */
300
301 * <p>
302 * A -e argument causes the input to be echoed.
303 * This can be useful for executing the program against
304 * a test script, e.g.,
305 * <pre>
306 * java Bank -e < Bank.in
307 * </pre>
308 *
309 * @param args the command line arguments:
310 *     <pre>
311 *     -e echo input.
312 *     bankName any other command line argument.
313 *     </pre>
314 */
315
316 public static void main( String[] args )
317 {
318     // parse the command line arguments for the echo
319     // flag and the name of the bank
320
321     boolean echo = false; // default does not echo
322     String bankName = "Falthless Trust"; // default bank name
323
324     for (int i = 0; i < args.length; i++) {
325         if (args[i].equals("-e")) {
326             echo = true;
327         }
328         else {
329             bankName = args[i];
330         }
331     }
332     Bank aBank = new Bank( bankName, new Terminal(echo) );
333     aBank.visit();
334 }
335

```

```

1 // fo1/5/bank/BankAccount.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A BankAccount object has private fields to keep track
8  * of its current balance, the number of transactions
9  * performed and the Bank in which it is an account, and
10 * and public methods to access those fields appropriately.
11 *
12 * @see Bank
13 * @version 5
14 */
15
16 public abstract class BankAccount
17 {
18     private int balance = 0; // Account balance (whole dollars)
19     private int transactionCount = 0; // Number of transactions performed
20     private Bank issuingBank; // Bank issuing this account
21
22     /**
23      * Construct a BankAccount with the given initial balance and
24      * issuing Bank. Construction counts as this BankAccount's
25      * first transaction.
26      *
27      * @param initialBalance the opening balance.
28      * @param issuingBank the bank that issued this account.
29      */
30
31     public BankAccount( int initialBalance, Bank issuingBank )
32     {
33         this.issuingBank = issuingBank;
34         deposit( initialBalance );
35     }
36
37     /**
38      * Withdraw the given amount, decreasing this BankAccount's
39      * balance and the issuing Bank's balance.
40      * Counts as a transaction.
41      *
42      * @param amount the amount to be withdrawn
43      * @return amount withdrawn
44      */
45
46     public int withdraw( int amount )
47     {
48         incrementBalance( -amount );
49         countTransaction();
50         return amount ;
51     }
52
53     /**
54      * Deposit the given amount, increasing this BankAccount's
55      * balance and the issuing Bank's balance.
56      * Counts as a transaction.

```

```

57     *
58     * @param amount the amount to be deposited
59     * @return amount deposited
60     */
61
62     public int deposit( int amount )
63     {
64         incrementBalance( amount );
65         countTransaction();
66         return amount ;
67     }
68
69     /**
70      * Request for balance. Counts as a transaction.
71      *
72      * @return current account balance.
73      */
74
75     public int requestBalance()
76     {
77         countTransaction();
78         return getBalance() ;
79     }
80
81     /**
82      * Get the current balance.
83      * Does NOT count as a transaction.
84      *
85      * @return current account balance
86      */
87
88     public int getBalance()
89     {
90         return balance;
91     }
92
93     /**
94      * Increment account balance by given amount.
95      * Also increment issuing Bank's balance.
96      * Does NOT count as a transaction.
97      *
98      * @param amount the amount of the increment.
99      */
100
101     public void incrementBalance( int amount )
102     {
103         balance += amount;
104         this.getIssuingBank().incrementBalance( amount );
105     }
106
107     /**
108      * Get the number of transactions performed by this
109      * account. Does NOT count as a transaction.
110      *
111      * @return number of transactions performed.
112     */

```

```
113 public int getTransactionCount()
114 {
115     return transactionCount;
116 }
117
118 /**
119  * Increment by 1 the count of transactions, for this account
120  * and for the issuing Bank.
121  * Does NOT count as a transaction.
122  */
123
124 public void countTransaction()
125 {
126     transactionCount++;
127     this.getIssuingBank().countTransaction();
128 }
129
130 /**
131  * Get the bank that issued this account.
132  * Does NOT count as a transaction.
133  * @return issuing bank.
134  */
135
136 public Bank getIssuingBank()
137 {
138     return issuingBank;
139 }
140
141 /**
142  * Action to take when a new month starts.
143  */
144
145 public abstract void newMonth();
146
147 }
148 }
```

```
1 // fo1/5/bank/RegularAccount.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A RegularAccount is a BankAccount that has no special behavior.
8  *
9  * It does what a BankAccount does.
10 */
11
12 public class RegularAccount extends BankAccount
13 {
14
15     /**
16     * Construct a BankAccount with the given initial balance and
17     * issuing Bank. Construction counts as this BankAccount's
18     * first transaction.
19     *
20     * @param initialBalance the opening balance.
21     * @param issuingBank the bank that issued this account.
22     */
23
24     public RegularAccount( int initialBalance, Bank issuingBank )
25     {
26         super( initialBalance, issuingBank );
27     }
28
29     /**
30     * Action to take when a new month starts.
31     *
32     * A RegularAccount does nothing when the next month starts.
33     */
34
35     public void newMonth() {
36         // do nothing
37     }
38
39 }
```



```
1 // fo1/5/bank/CheckingAccount.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A CheckingAccount is a BankAccount with one new feature:
8  * the ability to cash a check by calling the honorCheck method.
9  * Each honored check costs the customer a checkFee.
10 *
11 * @version 5
12 */
13
14 public class CheckingAccount extends BankAccount
15 {
16     private static int checkFee = 2; // pretty steep for each check
17
18     /**
19      * Constructs a CheckingAccount with the given
20      * initial balance and issuing Bank.
21      * Counts as this account's first transaction.
22      */
23     * @param initialBalance the opening balance for this account.
24     * @param issuingBank the bank that issued this account.
25     */
26
27     public CheckingAccount( int initialBalance, Bank issuingBank )
28     {
29         super( initialBalance, issuingBank );
30     }
31
32     /**
33      * Honor a check:
34      * Charge the account the appropriate fee
35      * and withdraw the amount.
36      */
37     * @param amount amount (in whole dollars) to be withdrawn.
38     * @return the amount withdrawn.
39     */
40
41     public int honorCheck( int amount )
42     {
43         incrementBalance( - checkFee );
44         return withdraw( amount );
45     }
46
47     /**
48      * Action to take when a new month starts.
49      */
50
51     public void newMonth()
52     {
53     }
54 }
```

```
1 // fo1/5/bank/FeeAccount.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A FeeAccount is a BankAccount with one new feature:
8  * the user is charged for each transaction.
9  *
10 * @version 5
11 */
12
13 public class FeeAccount extends BankAccount
14 {
15     private static int transactionFee = 1;
16
17     /**
18      * Constructor, accepting an initial balance and issuing Bank.
19      *
20      * @param initialBalance the opening balance.
21      * @param issuingBank the bank that issued this account.
22      */
23
24     public FeeAccount( int initialBalance, Bank issuingBank )
25     {
26         super( initialBalance, issuingBank);
27     }
28
29     /**
30      * The way a transaction is counted for a FeeAccount: it levies
31      * a transaction fee as well as counting the transaction.
32      */
33
34     public void countTransaction()
35     {
36         incrementBalance( - transactionFee );
37         super.countTransaction();
38     }
39
40     /**
41      * Action to take when a new month starts.
42      */
43
44     public void newMonth()
45     {
46     }
47 }
```

```

1 // foj/5/bank/class Month
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.io.*;
7 import java.util.Calendar;
8
9 /**
10  * The Month class implements an object that keeps
11  * track of the month of the year.
12  *
13  * @version 5
14  */
15
16 public class Month
17 {
18     private static final String[] monthName =
19         {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
20          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
21
22     private int month;
23     private int year;
24
25     /**
26      * Month constructor constructs a Month object
27      * initialized to the current month and year.
28      */
29
30     public Month()
31     {
32         Calendar rightNow = Calendar.getInstance();
33         month = rightNow.get( Calendar.MONTH );
34         year = rightNow.get( Calendar.YEAR );
35     }
36
37     /**
38      * Advance to next month.
39      */
40
41     public void next()
42     {
43         // needs completion
44     }
45
46     /**
47      * How a Month is displayed as a String -
48      * for example, "Jan, 2003".
49      */
50     * @return String representation of the month.
51     */
52
53     //
54     //
55     //
56     {
57         public String toString()

```

```

57     /**
58      * For unit testing.
59      */
60
61     public static void main( String[] args )
62     {
63         Month m = new Month();
64         for (int i=0; i < 14; i++, m.next()) {
65             System.out.println(m);
66         }
67         for (int i=0; i < 35; i++, m.next()); // no loop body
68         System.out.println("three years later: " + m);
69         for (int i=0; i < 120; i++, m.next()); // no loop body
70         System.out.println("ten years later: " + m);
71     }
72 }

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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

```

1 // fo1/6/juno/Shell.java
2 //
3 //
4 // Copyright 2003, Ethan Bolker and Bill Campbell
5
6 import java.util.*;
7
8 /**
9  * Models a shell (command interpreter)
10  *
11  * The Shell knows the (Juno) system it's working in,
12  * the User who started it,
13  * and the console to which to send output.
14  *
15  * It keeps track of the the current working directory ( . ) .
16  *
17  * @version 6
18  */
19
20 public class Shell
21 {
22     private Juno system; // the operating system object
23     private User user; // the user logged in
24     private Terminal console; // the console for this shell
25     private Directory dot; // the current working directory
26
27     /**
28      * Construct a login shell for the given user and console.
29      *
30      * @param system a reference to the Juno system.
31      * @param user the User logging in.
32      * @param console a Terminal for input and output.
33      */
34
35     public Shell( Juno system, User user, Terminal console )
36     {
37         this.system = system;
38         this.user = user;
39         this.console = console;
40         dot = user.getHome(); // default current directory
41         CLIShell(); // start the command line interpreter
42     }
43
44     // Run the command line interpreter
45
46     private void CLIShell()
47     {
48         boolean moreWork = true;
49         while(moreWork) {
50             moreWork = interpret( console.readLine( getPrompt() ) );
51         }
52         console.println("goodbye");
53     }
54
55     // Interpret a String of the form
56     // shellcommand command-arguments

```

```

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

```



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

```

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

```

```

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

```

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

```
1 // fo1/6/juno/TypeCommand.java
2 //
3 //
4 // Copyright 2003, Ethan Bolker and Bill Campbell
5
6 import java.util.*;
7
8 /**
9  * The Juno shell command to display the contents of a
10  * text file.
11  * Usage:
12  * <pre>
13  *   type textfile
14  * </pre>
15  *
16  * @version 6
17  */
18
19 public class TypeCommand extends ShellCommand
20 {
21     /**
22      * Construct a TypeCommand object.
23      */
24
25     TypeCommand()
26     {
27         super( "display contents of a TextFile", "textfile" );
28     }
29
30     /**
31      * Display the contents of a TextFile.
32      *
33      * @param args the remainder of the command line.
34      * @param sh the current Shell
35      */
36
37     public void doIt( StringTokenizer args, Shell sh )
38     {
39         String filename = args.nextToken();
40         sh.getConsole().println(
41             ( (TextFile) sh.getDot() ).
42             retrieveFile( filename ) ).getContents() );
43     }
44 }
```

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

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

```

1 // fo1/6/juno/ShellCommandTable.java (version 6)
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * A ShellCommandTable object maintains a dispatch table of
10 * ShellCommand objects keyed by the command names used to invoke
11 * them.
12 *
13 * To add a new shell command to the table, install it from
14 * method fillTable().
15 *
16 * @see ShellCommand
17 *
18 * @version 6
19 */
20
21 public class ShellCommandTable
22 {
23     private Map table = new TreeMap();
24
25     /**
26      * Construct and fill a shell command table.
27      */
28
29     public ShellCommandTable()
30     {
31         fillTable();
32     }
33
34     /**
35      * Get a ShellCommand, given the command name key.
36      *
37      * @param key the name associated with the command we're
38      * looking for.
39      *
40      * @return the command we're looking for, null if none.
41      */
42
43     public ShellCommand lookup( String key )
44     {
45         return (ShellCommand)table.get( key );
46     }
47
48     /**
49      * Get an array of the command names.
50      *
51      * @return the array of command names.
52      */
53
54     public String[] getCommandNames()
55     {
56         return (String[]) table.keySet().toArray( new String[0] );

```

```

57     }
58
59     // Associate a command name with a ShellCommand.
60
61     private void install( String commandName, ShellCommand command )
62     {
63         table.put( commandName, command );
64     }
65
66     // Fill the dispatch table with ShellCommands, keyed by their
67     // command names.
68
69     private void fillTable()
70     {
71         install( "newfile", new NewFileCommand() );
72         install( "type", new TypeCommand() );
73         install( "mkdir", new MkdirCommand() );
74         install( "help", new HelpCommand() );
75     }
76 }

```

```

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

```

```

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

```



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

1 // fo1/6/juno/User.java
2 //
3 //
4 // Copyright 2003 Ethan Bolker and Bill Campbell
5
6 /**
7  * Model a Juno user. Each User has a login name,
8  * a home directory, and a real name.
9  *
10 * @version 6
11 */
12
13 public class User
14 {
15     private String name; // the User's login name
16     private Directory home; // her home Directory
17     private String realName; // her real name
18
19     /**
20      * Construct a new User.
21      *
22      * @param name the User's login name.
23      * @param home her home Directory.
24      * @param realName her real name.
25      */
26
27     public User( String name, Directory home, String realName )
28     {
29         this.name = name;
30         this.home = home;
31         this.realName = realName;
32     }
33
34     /**
35      * Get the User's login name.
36      *
37      * @return the name.
38      */
39
40     public String getName()
41     {
42         return name;
43     }
44
45     /**
46      * Convert the User to a String.
47      * The String representation for a User is her
48      * login name.
49      *
50      * @return the User's name.
51      */
52
53     public String toString()
54     {
55         return getName();
56     }

```

```

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

```

```

1 // fo1/7/bank/Bank.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * A Bank object simulates the behavior of a simple bank/ATM.
10  * It contains a Terminal object and a collection of
11  * BankAccount objects.
12  *
13  * The visit method opens this Bank for business,
14  * prompting the customer for input.
15  *
16  * To create a Bank and open it for business issue the command
17  * <code>java Bank</code>.
18  *
19  * @see BankAccount
20  * @version 7
21  */
22
23 public class Bank
24 {
25     private String bankName; // the name of this Bank
26     private Terminal atm; // for talking with the customer
27     private int balance = 0; // total cash on hand
28     private int transactionCount = 0; // number of Bank transactions
29     private Month month; // the current month.
30     private Map accountList; // mapping names to accounts.
31
32     private int checkFee = 2; // cost for each check
33     private int transactionFee = 1; // fee for each transaction
34     private int monthlyCharge = 5; // monthly charge
35     private double interestRate = 0.05; // annual rate paid on savings
36     private int maxRetrTransactions = 3; // for savings accounts
37
38     // what the banker can ask of the bank
39
40     private static final String BANKER_COMMANDS =
41         "Banker commands: " +
42         "exit, open, customer, nextmonth, report, help.";
43
44     // what the customer can ask of the bank
45
46     private static final String CUSTOMER_TRANSACTIONS =
47         "Customer transactions: " +
48         "deposit, withdraw, transfer, balance, cash check, quit, help.";
49
50     /**
51      * Construct a Bank with the given name and Terminal.
52      *
53      * @param bankName the name for this Bank.
54      * @param atm this Bank's Terminal.
55      */
56

```

```

57     public Bank( String bankName, Terminal atm )
58     {
59         this.atm = atm;
60         this.bankName = bankName;
61         accountList = new TreeMap();
62         month = new Month();
63     }
64
65     /**
66      * Simulates interaction with a Bank.
67      * Presents the user with an interactive loop, prompting for
68      * banker transactions and in the case of the banker
69      * transaction "customer", an account id and further
70      * customer transactions.
71      */
72     public void visit()
73     {
74         instructUser();
75         String command;
76         while ( ! (command =
77             atm.readWord("banker command:")).equals("exit") ) {
78
79             if (command.startsWith("h")) {
80                 help( BANKER_COMMANDS );
81             }
82             else if (command.startsWith("o")) {
83                 openNewAccount();
84             }
85             else if (command.startsWith("n")) {
86                 newMonth();
87             }
88             else if (command.startsWith("r")) {
89                 report();
90             }
91             else if (command.startsWith("c")) {
92                 BankAccount acct = whichAccount();
93                 processTransactionsForAccount( acct );
94             }
95             else if (command.startsWith("a")) {
96                 atm.println( "unknown command: " + command );
97             }
98             else {
99                 // Unrecognized Request
100                atm.println( "unknown command: " + command );
101            }
102        }
103        report();
104        atm.println( "Goodbye from " + bankName );
105    }
106
107    /**
108     * Open a new bank account,
109     * prompting the user for information.
110     */
111    private void openNewAccount()
112

```

```

113     {
114         String accountName = atm.readWord("Account name: ");
115         char accountType =
116             atm.readChar("Type of account (r/c/f/s): ");
117         try {
118             int startup = readPosAmt("Initial deposit: ");
119             BankAccount newAccount;
120             switch( accountType ) {
121                 case 'c':
122                     newAccount = new CheckingAccount(startup, this);
123                     break;
124                 case 'f':
125                     newAccount = new FeeAccount(startup, this);
126                     break;
127                 case 's':
128                     newAccount = new SavingsAccount(startup, this);
129                     break;
130                 case 'r':
131                     newAccount = new RegularAccount( startup, this );
132                     break;
133                 default:
134                     atm.println("invalid account type: " + accountType);
135                     return;
136             }
137             accountList.put( accountName, newAccount );
138             atm.println( "opened new account " + accountName
139                 + " with $" + startup );
140         } // end of try block
141         catch (NegativeAmountException e) {
142             atm.errPrintln(
143                 "can't start with a negative balance");
144         }
145         catch (InsufficientFundsException e) {
146             atm.errPrintln("Initial deposit less than fee");
147         }
148     }
149
150     // Prompt the customer for transaction to process.
151     // Then send an appropriate message to the account.
152
153     private void processTransactionsForAccount( BankAccount acct )
154     {
155         help( CUSTOMER_TRANSACTIONS );
156
157         String transaction;
158         while (!(transaction =
159             atm.readWord(" transaction: ")).equals("quit")) {
160
161             try {
162                 if ( transaction.startsWith( "h" ) ) {
163                     help( CUSTOMER_TRANSACTIONS );
164                 }
165                 else if ( transaction.startsWith( "d" ) ) {
166                     int amount = readPosAmt( " amount: " );
167                     atm.println( " deposited "
168                         + acct.deposit( amount ) );

```

```

169     }
170     else if ( transaction.startsWith( "w" ) ) {
171         int amount = readPosAmt( " amount: " );
172         atm.println( " withdrew "
173             + acct.withdraw( amount ) );
174     }
175     else if ( transaction.startsWith( "c" ) ) {
176         int amount = readPosAmt( " amount of check: " );
177         try { // to cast acct to CheckingAccount ...
178             atm.println( " cashed check for " +
179                 ((CheckingAccount) acct).honorCheck( amount ) )
180         }
181         catch (ClassCastException e) {
182             // if not a checking account, report error
183             atm.errPrintln(
184                 " Sorry, not a checking account. " );
185         }
186     }
187     else if (transaction.startsWith("t")) {
188         atm.print( " to ");
189         BankAccount toacct = whichAccount();
190         if (toacct != null) {
191             int amount = readPosAmt(" amount to transfer: ");
192             atm.println(" transferred "
193                 + toacct.deposit(acct.withdraw(amount)));
194         }
195     }
196     else if (transaction.startsWith("b")) {
197         atm.println(" current balance "
198             + acct.requestBalance());
199     }
200     else {
201         atm.println(" sorry, unknown transaction" );
202     }
203     }
204     catch (InsufficientFundsException e) {
205         atm.errPrintln( " Insufficient funds " +
206             e.getMessage() );
207     }
208     catch (NegativeAmountException e) {
209         atm.errPrintln(" Sorry, negative amounts disallowed. ");
210     }
211     atm.println();
212 }
213
214 // Prompt for an account name (or number), look it up
215 // in the account list. If it's there, return it;
216 // otherwise report an error and return null.
217
218 private BankAccount whichAccount()
219 {
220     String accountName = atm.readWord( "account name: " );
221     BankAccount account = (BankAccount) accountList.get(accountName);
222     if (account == null) {
223         atm.println( "not a valid account" );
224     }

```

```

225     }
226     return account;
227 }
228
229 // Action to take when a new month starts.
230 // Update the month field by sending a next message.
231 // Loop on all accounts, sending each a newMonth message.
232
233 private void newMonth()
234 {
235     month.next();
236     Iterator i = accountList.keySet().iterator();
237     while ( i.hasNext() ) {
238         String name = (String) i.next();
239         BankAccount acct = (BankAccount)accountList.get(name);
240         try {
241             acct.newMonth();
242         }
243         catch (InsufficientFundsException e) {
244             atm.errPrintln(
245                 "Insufficient funds in account \"" +
246                 name + "\" for monthly fee" );
247         }
248     }
249 }
250
251 // Report bank activity. For each BankAccount,
252 // print the customer id (name or number), balance, and
253 // the number of transactions. Then print Bank totals.
254
255 private void report()
256 {
257     atm.println( bankName + " report for " + month );
258     atm.println( "\nSummaries of individual accounts:" );
259     atm.println( "account balance transaction count" );
260     for ( Iterator i = accountList.keySet().iterator();
261           i.hasNext(); ) {
262         String accountName = (String) i.next();
263         BankAccount acct = (BankAccount) accountList.get(accountName)
264         atm.println(accountName + "\t$" + acct.getBalance() + "\t\t"
265             + acct.getTransactionCount());
266     }
267     atm.println( "\nBank totals" );
268     atm.println( "open accounts: " + getNumberOfAccounts() );
269     atm.println( "cash on hand: $" + getBalance() );
270     atm.println( "transactions: " + getTransactionCount() );
271     atm.println();
272 }
273
274 // Welcome the user to the bank and instruct her on
275 // her options.
276
277 private void instructUser()
278 {
279     atm.println( "Welcome to " + bankName );
280

```

```

281     atm.println( month.toString() );
282     atm.println( "Open some accounts and work with them." );
283     help( BANKER_COMMANDS );
284 }
285
286 // Display a help string.
287
288 private void help( String helpString )
289 {
290     atm.println( helpString );
291     atm.println();
292 }
293
294 // Read amount prompted for from the atm.
295 // Throw a NegativeAmountException if amount < 0
296
297 private int readPosAmt( String prompt )
298     throws NegativeAmountException
299 {
300     int amount = atm.readInt( prompt );
301     if (amount < 0) {
302         throw new NegativeAmountException();
303     }
304     return amount;
305 }
306
307 /**
308  * Increment bank balance by given amount.
309  */
310 * @param amount the amount increment.
311 */
312
313 public void incrementBalance(int amount)
314 {
315     balance += amount;
316 }
317
318 /**
319  * Increment by one the count of transactions,
320  * for this bank.
321  */
322
323 public void countTransaction()
324 {
325     transactionCount++;
326 }
327
328 /**
329  * Get the number of transactions performed by this bank.
330  */
331 * @return number of transactions performed.
332 */
333
334 public int getTransactionCount()
335 {
336     return transactionCount ;
337 }

```

```

337     }
338     /**
339     * The charge this bank levies for cashing a check.
340     */
341     * @return check fee
342     */
343     public int getCheckFee( )
344     {
345         return checkFee ;
346     }
347     /**
348     * The charge this bank levies for a transaction.
349     */
350     * @return the transaction fee
351     */
352     public int getTransactionFee( )
353     {
354         return transactionFee ;
355     }
356     /**
357     * The charge this bank levies each month.
358     */
359     * @return the monthly charge
360     */
361     public int getMonthlyCharge( )
362     {
363         return monthlyCharge;
364     }
365     /**
366     * The current interest rate on savings.
367     */
368     * @return the interest rate
369     */
370     public double getInterestRate( )
371     {
372         return interestRate;
373     }
374     /**
375     * The number of free transactions per month.
376     */
377     * @return the number of transactions
378     */
379     public int getMaxFreeTransactions( )
380     {
381         return maxFreeTransactions;
382     }
383     }
384 }
385 }
386 }
387 }
388 }
389 }
390 }
391 }
392 }

```

```

393     /**
394     * Get the current bank balance.
395     */
396     * @return current bank balance.
397     */
398     public int getBalance( )
399     {
400         return balance;
401     }
402     /**
403     * Get the current number of open accounts.
404     */
405     * @return number of open accounts.
406     */
407     public int getNumberOfAccounts( )
408     {
409         return accountList.size();
410     }
411     /**
412     * Run the simulation by creating and then visiting a new Bank.
413     * <p>
414     * A -e argument causes the input to be echoed.
415     * This can be useful for executing the program against
416     * a test script, e.g.,
417     * <pre>
418     * java Bank -e < Bank.in
419     * </pre>
420     * @param args the command line arguments:
421     *     <pre>
422     *     -e echo input.
423     *     bankName any other command line argument.
424     * </pre>
425     */
426     public static void main( String[] args )
427     {
428         // parse the command line arguments for the echo
429         // flag and the name of the bank
430         boolean echo = false;
431         String bankName = "River Bank"; // default does not echo
432         for (int i = 0; i < args.length; i++ ) {
433             if (args[i].equals("-e")) {
434                 echo = true;
435             }
436             else {
437                 bankName = args[i];
438             }
439         }
440     }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }

```



```
449     Bank aBank = new Bank( bankName, new Terminal(echo) );
450     }
451     aBank.visit();
452 }
```

```

1 // fo1/7/bank/BankAccount.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A BankAccount object has private fields to keep track
8  * of its current balance, the number of transactions
9  * performed and the Bank in which it is an account, and
10 * and public methods to access those fields appropriately.
11 *
12 * @see Bank
13 * @version 7
14 */
15
16 public abstract class BankAccount
17 {
18     private int balance = 0; // Account balance (whole dollars)
19     private int transactionCount = 0; // Number of transactions performed
20     private Bank issuingBank; // Bank issuing this account
21
22     /**
23      * Construct a BankAccount with the given initial balance and
24      * issuing Bank. Construction counts as this BankAccount's
25      * first transaction.
26      *
27      * @param initialBalance the opening balance.
28      * @param issuingBank the bank that issued this account.
29      *
30      * @exception InsufficientFundsException when appropriate.
31      */
32     protected BankAccount( int initialBalance, Bank issuingBank )
33     throws InsufficientFundsException
34     {
35         this.issuingBank = issuingBank;
36         deposit( initialBalance );
37     }
38
39     /**
40      * Get transaction fee. By default, 0.
41      *
42      * Override this for accounts having transaction fees.
43      *
44      * @return the fee.
45      */
46     protected int getTransactionFee()
47     {
48         return 0;
49     }
50
51     /**
52      * The bank that issued this account.
53      *
54      * @return the Bank.
55      */
56

```

```

57
58     protected Bank getIssuingBank()
59     {
60         return issuingBank;
61     }
62
63     /**
64      * Withdraw the given amount, decreasing this BankAccount's
65      * balance and the issuing Bank's balance.
66      * Counts as a transaction.
67      *
68      * @param amount the amount to be withdrawn
69      * @return amount withdrawn
70      *
71      * @exception InsufficientFundsException when appropriate.
72      */
73
74     public int withdraw( int amount )
75     throws InsufficientFundsException
76     {
77         incrementBalance( -amount - getTransactionFee() );
78         countTransaction();
79         return amount ;
80     }
81
82     /**
83      * Deposit the given amount, increasing this BankAccount's
84      * balance and the issuing Bank's balance.
85      * Counts as a transaction.
86      *
87      * @param amount the amount to be deposited
88      * @return amount deposited
89      *
90      * @exception InsufficientFundsException when appropriate.
91      */
92
93     public int deposit( int amount )
94     throws InsufficientFundsException
95     {
96         incrementBalance( amount - getTransactionFee() );
97         countTransaction();
98         return amount ;
99     }
100
101     /**
102      * Request for balance. Counts as a transaction.
103      *
104      * @return current account balance.
105      *
106      * @exception InsufficientFundsException when appropriate.
107      */
108
109     public int requestBalance()
110     throws InsufficientFundsException
111     {
112         incrementBalance( - getTransactionFee() );

```

```

113     countTransaction();
114     return getBalance() ;
115 }
116
117 /**
118  * Get the current balance.
119  * Does NOT count as a transaction.
120  */
121     @return current account balance
122     */
123     public int getBalance()
124     {
125         return balance;
126     }
127
128
129 /**
130  * Increment account balance by given amount.
131  * Also increment issuing Bank's balance.
132  * Does NOT count as a transaction.
133  */
134     @param amount the amount of the increment.
135     * @exception InsufficientFundsException when appropriate.
136     */
137
138
139     public final void incrementBalance( int amount )
140     {
141         throws InsufficientFundsException
142         int newBalance = balance + amount;
143         if (newBalance < 0) {
144             throw new InsufficientFundsException(
145                 "For this transaction" );
146         }
147         balance = newBalance;
148         getIssuingBank().incrementBalance( amount );
149     }
150
151 /**
152  * Get the number of transactions performed by this
153  * account. Does NOT count as a transaction.
154  */
155     * @return number of transactions performed.
156     */
157
158     public int getTransactionCount()
159     {
160         return transactionCount;
161     }
162
163 /**
164  * Increment by 1 the count of transactions, for this account
165  * and for the issuing Bank.
166  * Does NOT count as a transaction.
167  */
168     * @exception InsufficientFundsException when appropriate.

```

```

169     */
170
171     public void countTransaction()
172     {
173         throws InsufficientFundsException
174         transactionCount++;
175         this.getIssuingBank().countTransaction();
176     }
177
178 /**
179  * Action to take when a new month starts.
180  */
181     * @exception InsufficientFundsException thrown when funds
182     * on hand are not enough to cover the fees.
183     */
184
185     public abstract void newMonth()
186     {
187         throws InsufficientFundsException;
188     }

```

```

1 // fo1/7/bank/CheckingAccount.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A CheckingAccount is a BankAccount with one new feature:
8  * the ability to cash a check by calling the honorCheck method.
9  * Each honored check costs the customer a checkFee.
10 *
11 * @see BankAccount
12 *
13 * @version 7
14 */
15
16 public class CheckingAccount extends BankAccount
17 {
18     /**
19     * Constructs a CheckingAccount with the given
20     * initial balance and issuing Bank.
21     * Counts as this account's first transaction.
22     *
23     * @param initialBalance the opening balance for this account.
24     * @param issuingBank the bank that issued this account.
25     *
26     * @exception InsufficientFundsException when appropriate.
27     */
28
29     public CheckingAccount( int initialBalance, Bank issuingBank )
30     throws InsufficientFundsException
31     {
32         super( initialBalance, issuingBank );
33     }
34
35     /**
36     * Honor a check:
37     * Charge the account the appropriate fee
38     * and withdraw the amount.
39     *
40     * @param amount amount (in whole dollars) to be withdrawn.
41     * @return the amount withdrawn.
42     *
43     * @exception InsufficientFundsException when appropriate.
44     */
45
46     public int honorCheck( int amount )
47     throws InsufficientFundsException
48     {
49         // careful error checking logic:
50         // first try to deduct the check fee
51         // if you succeed, try to honor check
52         // if that fails, remember to add back the check fee!
53
54         try {
55             incrementBalance( - getIssuingBank().getCheckFee() );
56

```

```

57         catch (InsufficientFundsException e) {
58             throw new InsufficientFundsException(
59                 "to cover check fee" );
60         }
61         try {
62             withdraw( amount );
63         }
64         catch (InsufficientFundsException e) {
65             incrementBalance( getIssuingBank().getCheckFee() );
66             throw new InsufficientFundsException(
67                 "to cover check + check fee" );
68         }
69         return amount;
70     }
71
72     /**
73     * Nothing special happens to a CheckingAccount on the
74     * first day of the month.
75     */
76     public void newMonth()
77     {
78         return;
79     }
80
81 }

```

```

1 // fo1/7/bank/SavingsAccount.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A SavingsAccount is a BankAccount that bears interest.
8  * A fee is charged for too many transactions in a month.
9  *
10 * @see BankAccount
11 *
12 * @version 7
13 */
14
15 public class SavingsAccount extends BankAccount
16 {
17     private int transactionsThisMonth;
18
19     /**
20      * Override getTransactionFee() to return a non-zero fee
21      * after the appropriate number of free monthly transactions.
22      *
23      * @return the fee for current transaction.
24      */
25     protected int getTransactionFee()
26     {
27         if (transactionsThisMonth >
28             getIssuingBank().getMaxFreeTransactions()) {
29             return getIssuingBank().getTransactionFee();
30         }
31         else {
32             return 0;
33         }
34     }
35
36     /**
37      * Increment count of transactions, for this account for
38      * this Month and in total and for the issuing Bank, by one.
39      *
40      * @exception InsufficientFundsException when appropriate.
41      */
42     public void countTransaction()
43     throws InsufficientFundsException
44     {
45         transactionsThisMonth++;
46         super.countTransaction();
47     }
48
49     /**
50      * Constructor, accepting an initial balance.
51      * @param initialBalance the opening balance.
52      * @param issuingBank the bank that issued this account.
53      *
54      *
55      *
56

```

```

57     * @exception InsufficientFundsException when appropriate.
58     */
59     public SavingsAccount( int initialBalance, Bank issuingBank )
60     throws InsufficientFundsException
61     {
62         super( initialBalance, issuingBank);
63         transactionsThisMonth = 1;
64     }
65
66     /**
67      * A SavingsAccount earns interest each month.
68      *
69      * @exception InsufficientFundsException when appropriate.
70      */
71     public void newMonth()
72     throws InsufficientFundsException
73     {
74         double monthlyRate = getIssuingBank().getInterestRate()/12;
75         incrementBalance( (int)(monthlyRate * getBalance()));
76         transactionsThisMonth = 0;
77     }
78
79     }
80

```

```

1 // fo1/7/bank/FeeAccount.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A FeeAccount is a BankAccount with one new feature:
8  * the user is charged for each transaction.
9  *
10 * @see BankAccount
11 *
12 * @version 7
13 */
14
15 public class FeeAccount extends BankAccount
16 {
17     /**
18     * Constructor, accepting an initial balance and issuing Bank.
19     *
20     * @param initialBalance the opening balance.
21     * @param issuingBank the bank that issued this account.
22     *
23     * @exception InsufficientFundsException when appropriate.
24     */
25
26     public FeeAccount( int initialBalance, Bank issuingBank )
27     throws InsufficientFundsException
28     {
29         super( initialBalance, issuingBank);
30     }
31
32     /**
33     * The Bank's transaction fee.
34     *
35     * @return the fee.
36     */
37
38     protected int getTransactionFee()
39     {
40         return getIssuingBank().getTransactionFee();
41     }
42
43     /**
44     * The way a transaction is counted for a FeeAccount: it levies
45     * a transaction fee as well as counting the transaction.
46     *
47     * @exception InsufficientFundsException when appropriate.
48     */
49
50     public void countTransaction()
51     throws InsufficientFundsException
52     {
53         incrementBalance( - getTransactionFee() );
54         super.countTransaction();
55     }
56

```

```

57     /**
58     * A FeeAccount incurs a monthly charge.
59     *
60     * @exception InsufficientFundsException when appropriate.
61     */
62
63     public void newMonth()
64     throws InsufficientFundsException
65     {
66         incrementBalance( - getIssuingBank().getMonthlyCharge());
67     }
68 }

```

```
1 // fo1/5/bank/RegularAccount.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A RegularAccount is a BankAccount that has no special behavior.
8  *
9  * It does what a BankAccount does.
10 */
11
12 public class RegularAccount extends BankAccount
13 {
14
15     /**
16     * Construct a BankAccount with the given initial balance and
17     * issuing Bank. Construction counts as this BankAccount's
18     * first transaction.
19     *
20     * @param initialBalance the opening balance.
21     * @param issuingBank the bank that issued this account.
22     *
23     * @exception InsufficientFundsException when appropriate.
24     */
25
26     public RegularAccount( int initialBalance, Bank issuingBank )
27     throws InsufficientFundsException
28     {
29         super( initialBalance, issuingBank );
30     }
31
32     /**
33     * Action to take when a new month starts.
34     *
35     * A RegularAccount does nothing when the next month starts.
36     */
37
38     public void newMonth() {
39         // do nothing
40     }
41
42 }
```

```

1 // foj/7/bank/Class Month
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.io.*;
7 import java.util.Calendar;
8
9 /**
10 * The Month class implements an object that keeps
11 * track of the month of the year.
12 *
13 * @version 7
14 */
15
16 public class Month
17 {
18     private static final String[] monthName =
19         {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
20          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
21
22     private int month;
23     private int year;
24
25     /**
26      * Month constructor constructs a Month object
27      * initialized to the current month and year.
28      */
29
30     public Month()
31     {
32         Calendar rightNow = Calendar.getInstance();
33         month = rightNow.get( Calendar.MONTH );
34         year = rightNow.get( Calendar.YEAR );
35     }
36
37     /**
38      * Advance to next month.
39      */
40
41     public void next()
42     {
43         month = (month + 1) % 12;
44         if (month == 0) {
45             year++;
46         }
47     }
48
49     /**
50      * How a Month is displayed as a String -
51      * for example, "Jan, 2003".
52      */
53     * @return String representation of the month.
54     */
55     public String toString()

```

```

57     {
58         return monthName[month] + ", " + year;
59     }
60
61     /**
62      * For unit testing.
63      */
64
65     public static void main( String[] args )
66     {
67         Month m = new Month();
68         for (int i=0; i < 14; i++, m.next()) {
69             System.out.println(m);
70         }
71         for (int i=0; i < 35; i++, m.next()) { // no loop body
72             System.out.println( "three years later: " + m );
73             for (int i=0; i < 120; i++, m.next()) { // no loop body
74                 System.out.println( "ten years later: " + m );
75             }
76         }

```



```
1 // fo1/7/bank/InsufficientFundsException.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * Thrown when there is an attempt to spend money that is not there.
8  *
9  * @version 7
10 */
11
12 public class InsufficientFundsException extends Exception
13 {
14     /**
15      * Construct an InsufficientFundsException
16      * with a String description.
17      *
18      * @param msg a more specific description.
19      */
20
21     public InsufficientFundsException( String msg )
22     {
23         super( msg );
24     }
25
26     /**
27      * Construct an InsufficientFundsException
28      * with no description.
29      */
30
31     public InsufficientFundsException()
32     {
33         this( "" );
34     }
35 }
```

```
1 // foj/7/bank/NegativeAmountException.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * Thrown when attempting to work with a negative amount.
8  *
9  * @version 7
10 */
11
12 public class NegativeAmountException extends Exception
13 {
14 }
```

```

1 // fo1/7/juno/Juno.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.io.*;
7 import java.util.*;
8 import java.lang.*;
9
10 /**
11  * Juno (Juno's Unix NOC) mimics a command line operating system
12  * like Unix.
13  * <p>
14  * A Juno system has a name, a set of Users, a JFile system,
15  * a login process and a set of shell commands.
16  *
17  * @see User
18  * @see JFile
19  * @see ShellCommand
20  *
21  * @version 7
22  */
23
24 public class Juno
25 {
26     private final static String OS      = "Juno";
27     private final static String VERSION = "7";
28
29     private String  hostName; // host machine name
30     private Map    users; // lookup table for Users
31     private Terminal console; // for input and output
32
33     private Directory slash; // root of JFile system
34     private Directory userHomes; // for home directories
35
36     private ShellCommandTable commandTable; // shell commands
37
38     /**
39      * Construct a Juno (operating system) object.
40      *
41      * @param hostName the name of the host on which it's running.
42      * @param echoInput should all input be echoed as output?
43      */
44
45     public Juno( String hostName, boolean echoInput )
46     {
47         // initialize the Juno environment ...
48
49         this.hostName = hostName;
50         console       = new Terminal( echoInput );
51         users         = new TreeMap(); // for registered Users
52         commandTable = new ShellCommandTable(); // for shell commands
53
54         // the file system
55         slash = new Directory( "", null, null );
56

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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

```

1 // fo1/7/juno/Shell.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * Models a shell (command interpreter)
10  *
11  * The Shell knows the (Juno) system it's working in,
12  * the User who started it,
13  * and the console to which to send output.
14
15  * It keeps track of the the current working directory ( . ) .
16  *
17  * @version 7
18  */
19
20 public class Shell
21 {
22     private Juno system; // the operating system object
23     private User user; // the user logged in
24     private Terminal console; // the console for this shell
25     private Directory dot; // the current working directory
26
27     /**
28      * Construct a login shell for the given user and console.
29      *
30      * @param system a reference to the Juno system.
31      * @param user the User logging in.
32      * @param console a Terminal for input and output.
33      */
34
35     public Shell( Juno system, User user, Terminal console )
36     {
37         this.system = system;
38         this.user = user;
39         this.console = console;
40         dot = user.getHome(); // default current directory
41         CLIShell();
42     }
43
44     // Run the command line interpreter
45
46     private void CLIShell()
47     {
48         boolean moreWork = true;
49         while(moreWork) {
50             moreWork = interpret( console.readLine( getPrompt() ) );
51         }
52         console.println("goodbye");
53     }
54
55     // Interpret a String of the form
56     // shellcommand command-arguments

```

```

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

```

```
113     } return system.getHostHostName() + "> ";
114
115     /**
116     * The User associated with this shell.
117     * @return the user.
118     */
119     public User getUser()
120     {
121     }
122
123     /**
124     * The current working directory for this shell.
125     * @return the current working directory.
126     */
127     public Directory getDot()
128     {
129     }
130
131     /**
132     * Set the current working directory for this shell.
133     * @param dot the new working directory.
134     */
135     public void setDot(Directory dot)
136     {
137     this.dot = dot;
138     }
139
140     /**
141     * The console associated with this Shell.
142     * @return the console.
143     */
144     public Terminal getConsole()
145     {
146     return console;
147     }
148
149     /**
150     * The Juno object associated with this Shell.
151     * @return the Juno instance that created this Shell.
152     */
153     public Juno getSystem()
154     {
155     return system;
156     }
157
158     /**
159     * The Juno object associated with this Shell.
160     * @return the Juno instance that created this Shell.
161     */
162     public Juno getSystem()
163     {
164     return system;
165     }
166
167     /**
168     * The Juno object associated with this Shell.
169     * @return the Juno instance that created this Shell.
170     */
170     public Juno getSystem()
171     {
172     return system;
173     }
```

```
169     }
170 }
```



```

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

```

```

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

```

```

1 // fo1/7/juno/ShellCommandTable.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * A ShellCommandTable object maintains a dispatch table of
10 * ShellCommand objects keyed by the command names used to invoke
11 * them.
12 *
13 * To add a new shell command to the table, install it from
14 * method fillTable().
15 *
16 * @see ShellCommand
17 *
18 * @version 7
19 */
20
21 public class ShellCommandTable
22 {
23     private Map table = new TreeMap();
24
25     /**
26      * Construct and fill a shell command table.
27      */
28
29     public ShellCommandTable()
30     {
31         fillTable();
32     }
33
34     /**
35      * Get a ShellCommand, given the command name key.
36      *
37      * @param key the name associated with the command we're
38      * looking for.
39      *
40      * @return the command we're looking for, null if none.
41      */
42
43     public ShellCommand lookup( String key )
44     {
45         ShellCommand commandObject = (ShellCommand) table.get( key );
46         if (commandObject != null) {
47             return commandObject;
48         }
49
50         // try to load dynamically
51         // construct classname = "KeyCommand"
52         char[] chars = (key + "Command").toCharArray();
53         chars[0] = key.toUpperCase().charAt(0);
54         String classname = new String(chars);
55         try {
56             commandObject =

```

```

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

```

```
1 // fo1/7/juno/MkdirCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * The Juno shell command to create a new directory.
10  * Usage:
11  * <pre>
12  *   mkdir directory-name
13  * </pre>
14  *
15  * @version 7
16  */
17
18 public class MkdirCommand extends ShellCommand
19 {
20     MkdirCommand()
21     {
22         super( "create a subdirectory of the current directory",
23             "directory-name" );
24     }
25
26     /**
27      * Create a new Directory in the current Directory.
28      *
29      * @param args the remainder of the command line.
30      * @param sh the current shell.
31      *
32      * @exception JunoException for reporting errors.
33      */
34
35     public void doit( StringTokenizer args, Shell sh )
36     {
37         throws JunoException
38     {
39         String filename = args.nextToken();
40         new Directory( filename, sh.getUser(), sh.getDot() );
41     }
42 }
```

```

1 // fo1/7/juno/TypeCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * The Juno shell command to display the contents of a
10 * text file.
11 * Usage:
12 * <pre>
13 *   type textfile
14 * </pre>
15 *
16 * @version 7
17 */
18
19 public class TypeCommand extends ShellCommand
20 {
21     TypeCommand()
22     {
23         super( "display contents of a TextFile", "textfile" );
24     }
25
26     /**
27     * Display the contents of a TextFile.
28     *
29     * @param args the remainder of the command line.
30     * @param sh the current Shell
31     *
32     * @exception JunoException for reporting errors
33     */
34
35     public void doit( StringTokenizer args, Shell sh )
36     throws JunoException
37     {
38         String filename;
39
40         try {
41             filename = args.nextToken();
42         }
43         catch (NoSuchElementException e) {
44             throw new BadShellCommandException( this );
45         }
46         try {
47             sh.getConsole().println(
48                 ( (TextFile) sh.getDot() ).
49                 retrieveFile( filename ) ).getContents();
50         }
51         catch (NullPointerException e) {
52             throw new JunoException( "JFile does not exist: "
53                 + filename);
54         }
55         catch (ClassCastException e) {
56             throw new JunoException( "JFile not a text file: "
57                 + filename);
58         }
59     }
60
61     // EEE
62 }

```

```

57     }
58 }
59 }

```

```

// EEE

```

```
1 // fo1/7/juno/HelpCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * The Juno shell command to display help on the shell commands.
10  * Usage:
11  * <pre>
12  *     help
13  * </pre>
14  *
15  * @version 7
16  */
17
18 public class HelpCommand extends ShellCommand
19 {
20     HelpCommand()
21     {
22         super( "display ShellCommands" );
23     }
24
25     /**
26     * Print out help for all commands.
27     *
28     * @param args the remainder of the command line.
29     * @param sh the current shell
30     *
31     * @exception JunoException for reporting errors
32     */
33
34     public void doIt( StringTokenizer args, Shell sh )
35     {
36         throws JunoException
37     {
38         // Get command keys from global table, print them out.
39
40         sh.getConsole().println( "shell commands" );
41         ShellCommandTable table = sh.getSystem().getCommandTable();
42         String[] names = table.getCommandNames();
43         for (int i = 0; i < names.length; i++) {
44             String cmdname = names[i];
45             ShellCommand cmd =
46                 (ShellCommand) table.lookup( cmdname );
47             sh.getConsole().
48                 println( " " + cmdname + " : " + cmd.getHelpString() );
49         }
50     }
51 }
```



```

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

```

```

57         }
58     }
59 }

```

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



```
1 // fo1/7/juno/LogoutCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * The Juno shell command to log out.
10  * Usage:
11  * <pre>
12  *   logout
13  * </pre>
14  *
15  * @version 7
16  */
17
18 public class LogoutCommand extends ShellCommand
19 {
20     LogoutCommand()
21     {
22         super( "log out, return to login: prompt" );
23     }
24
25     /**
26      * Log out from the current shell.
27      *
28      * @param args the remainder of the command line.
29      * @param sh the current shell
30      *
31      * @exception JunoException for reporting errors
32      */
33
34     public void doIt( StringTokenizer args, Shell sh )
35     {
36         throws JunoException
37     {
38         throw new ExitShellException();
39     }
39 }
```

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

1 // fo1/7/juno/User.java
2 //
3 //
4 // Copyright 2003 Ethan Bolker and Bill Campbell
5
6 /**
7  * Model a Juno user. Each User has a login name,
8  * a home directory, and a real name.
9  *
10 * @version 7
11 */
12
13 public class User
14 {
15     private String name; // the User's login name
16     private Directory home; // her home Directory
17     private String realName; // her real name
18
19     /**
20      * Construct a new User.
21      *
22      * @param name the User's login name.
23      * @param home her home Directory.
24      * @param realName her real name.
25      */
26
27     public User( String name, Directory home, String realName )
28     {
29         this.name = name;
30         this.home = home;
31         this.realName = realName;
32     }
33
34     /**
35      * Get the User's login name.
36      *
37      * @return the name.
38      */
39
40     public String getName()
41     {
42         return name;
43     }
44
45     /**
46      * Convert the User to a String.
47      * The String representation for a User is her
48      * login name.
49      *
50      * @return the User's name.
51      */
52
53     public String toString()
54     {
55         return getName();
56     }

```

```

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

```

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



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

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

```

1 // foj/8/terminal/Terminal.java
2 // (and terminal/Terminal.java)
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.io.*;
7
8 /**
9  * Terminal provides a user-friendly interface to the standard System
10 * input and output streams (in, out, and err).
11 * <p>
12 * A Terminal is an object. In general, one is expected to instantiate
13 * just one Terminal. Although one might instantiate several, all will
14 * share the same System streams.
15 * <p>
16 * A Terminal may either explicitly echo input, or not. Echoing input
17 * is useful, for example, when testing with I/O redirection.
18 * <p>
19 * Inspired by Cay Horstmann's Console Class.
20 */
21
22 public class Terminal
23 {
24     private boolean echo = false;
25     private static BufferedReader in =
26         new BufferedReader(new FileReader(FileDescriptor.in));
27
28
29     // Print a prompt to the console without a newline.
30
31     private void printPrompt( String prompt )
32     {
33         print( prompt );
34         System.out.flush();
35     }
36
37     /**
38      * Construct a Terminal that doesn't echo input.
39      */
40
41     public Terminal()
42     {
43         this( false );
44     }
45
46     /**
47      * Construct a Terminal.
48      *
49      * @param echo whether or not input should be echoed.
50      */
51
52     public Terminal( boolean echo )
53     {
54         this.echo = echo;
55     }
56

```

```

57
58     /**
59      * Read a line (terminated by a newline) from the Terminal.
60      * @param prompt output string to prompt for input.
61      * @return the string (without the newline character),
62      *         * null if eof.
63      */
64
65     public String readline( String prompt )
66     {
67         printPrompt(prompt);
68         try {
69             String line = in.readLine();
70             if (echo) {
71                 println(line);
72             }
73             return line;
74         }
75         catch (IOException e) {
76             return null;
77         }
78     }
79
80     /**
81      * Read a line (terminated by a newline) from the Terminal.
82      *
83      * @return the string (without the newline character).
84      */
85
86     public String readline()
87     {
88         return readline( "" );
89     }
90
91     // Read a line from the Terminal. An end of file,
92     // indicated by a null, raises a runtime exception.
93     // Used only internally.
94
95     private String readNonNullLine()
96     {
97         return readNonNullLine( "" );
98     }
99
100    // Read a line from the Terminal. An end of file,
101    // indicated by a null, raises a runtime exception.
102    // Used only internally.
103
104    private String readNonNullLine( String prompt )
105    {
106        String line = readline( prompt );
107        if (line == null) {
108            throw new RuntimeException( "End of file encountered." );
109        }
110        return line;
111    }
112

```

```

113  /**
114  * Read a word from the Terminal.
115  * If an empty line is entered, try again.
116  * Words are terminated by whitespace.
117  * Leading whitespace is trimmed; the rest of the line
118  * is disposed of.
119  *
120  * @param prompt output string to prompt for input.
121  * @return the word read.
122  */
123  public String readWord( String prompt )
124  {
125      String line = readNonNullLine( prompt );
126      if (line.length() == 0) {
127          println( "Empty line. Please try again." );
128          return readWord( "" );
129      }
130      line = line.trim();
131      for ( int i = 0; i < line.length(); i++ ) {
132          if ( Character.isWhitespace( line.charAt(i) ) ) {
133              return line.substring( 0, i );
134          }
135      }
136      return line;
137  }
138  /**
139  * Read a word from the Terminal.
140  * If an empty line is entered, try again.
141  * Words are terminated by whitespace.
142  * Leading whitespace is trimmed; the rest of the line
143  * is disposed of.
144  *
145  * @return the word read.
146  */
147  public String readWord()
148  {
149      return readWord( "" );
150  }
151  /**
152  * Read a word from the Terminal.
153  * If an empty line is entered, throw an exception.
154  * Words are terminated by whitespace.
155  * Leading whitespace is trimmed; the rest of the line
156  * is disposed of.
157  *
158  * @param prompt output string to prompt for input.
159  * @return the word read.
160  * @throws RuntimeException if it reads an empty line.
161  */
162  public String readWordOnce( String prompt )
163  {
164      String line = readNonNullLine( prompt );
165      if (line.length() == 0) {
166          println( "No character on line. Please try again." );
167          return readChar( "" );
168      }

```

```

169      return line;
170  }
171  /**
172  * Read a character from the Terminal.
173  * If an empty line is entered, throw an exception.
174  * Words are terminated by whitespace.
175  * Leading whitespace is trimmed; the rest of the line
176  * is disposed of.
177  *
178  * @param prompt output string to prompt for input.
179  * @return the character read.
180  */
181  public char readChar( String prompt )
182  {
183      String line = readNonNullLine( prompt );
184      if (line.length() == 0) {
185          println( "Empty line encountered." );
186          return readChar( "" );
187      }
188      line = line.trim();
189      for ( int i = 0; i < line.length(); i++ ) {
190          if ( Character.isWhitespace( line.charAt(i) ) ) {
191              return line.charAt( i );
192          }
193      }
194      return line.charAt( 0 );
195  }
196  /**
197  * Read a character from the Terminal.
198  * If an empty line is entered, throw an exception.
199  * Words are terminated by whitespace.
200  * Leading whitespace is trimmed; the rest of the line
201  * is disposed of.
202  *
203  * @return the character read.
204  * @throws RuntimeException if it reads an empty line.
205  */
206  public char readCharOnce( String prompt )
207  {
208      String line = readNonNullLine( prompt );
209      if (line.length() == 0) {
210          println( "No character on line. Please try again." );
211          return readChar( "" );
212      }
213      line = line.trim();
214      for ( int i = 0; i < line.length(); i++ ) {
215          if ( Character.isWhitespace( line.charAt(i) ) ) {
216              return line.charAt( i );
217          }
218      }
219      return line.charAt( 0 );
220  }
221  /**
222  * Read a character from the Terminal.
223  * If an empty line is entered, throw an exception.
224  * Words are terminated by whitespace.
225  * Leading whitespace is trimmed; the rest of the line
226  * is disposed of.
227  *
228  * @param prompt output string to prompt for input.
229  */
230  public char readChar()
231  {
232      return readChar( "" );
233  }
234  /**
235  * Read a character from the Terminal.
236  * If an empty line is entered, throw an exception.
237  * Words are terminated by whitespace.
238  * Leading whitespace is trimmed; the rest of the line
239  * is disposed of.
240  *
241  * @return the character read.
242  */
243  public char readCharOnce()
244  {
245      return readChar( "" );
246  }

```

```

225  * @return the character read.
226  * @throws RuntimeException if it reads an empty line.
227  */
228  * /
229  public char readCharOnce( String prompt )
230  {
231      String line = readNonNullLine(prompt);
232      if (line.length() == 0) {
233          throw new RuntimeException("Empty line encountered.");
234      }
235      return line.charAt(0);
236  }
237  }
238  }
239  }
240  /**
241  * Read a character from the Terminal.
242  * Prompt again when an empty line is read.
243  *
244  *
245  * @return the character read.
246  */
247  public char readChar()
248  {
249      return readChar("");
250  }
251  }
252  }
253  /**
254  * Read a character from the Terminal.
255  * Throw an exception if an empty line is read.
256  *
257  * @return the character read.
258  *
259  * @throws RuntimeException if it reads an empty line.
260  */
261  public char readCharOnce()
262  {
263      return readCharOnce("");
264  }
265  }
266  }
267  }
268  /**
269  * Read "yes" or "no" from the Terminal.
270  * If an empty line or improper character is read,
271  * try again.
272  * Look only at first character and accept any case.
273  *
274  * @param prompt output string to prompt for input.
275  * @return true if yes, false if no.
276  */
277  public boolean readYesOrNo( String prompt )
278  {
279     printPrompt( prompt );
280     while ( true ) {

```

```

281     char answer = readChar( " (y or n): " );
282     if ( answer == 'y' || answer == 'Y' ) {
283         return true;
284     }
285     else if ( answer == 'n' || answer == 'N' ) {
286         return false;
287     }
288     else {
289         printPrompt( "oops!" );
290     }
291 }
292 }
293 }
294 }
295 /**
296 * Read "yes" or "no" from the Terminal.
297 * If an empty line or improper character is read,
298 * throw an exception.
299 * Look only at first character and accept any case.
300 *
301 * @param prompt output string to prompt for input.
302 * @return true if yes, false if no.
303 *
304 * @throws RuntimeException on improper input.
305 */
306 public boolean readYesOrNoOnce( String prompt )
307 {
308     printPrompt( prompt );
309     while ( true ) {
310         char answer = readCharOnce( " (y or n): " );
311         if ( answer == 'y' || answer == 'Y' ) {
312             return true;
313         }
314         else if ( answer == 'n' || answer == 'N' ) {
315             return false;
316         }
317         else {
318             throw new RuntimeException( "Must be y or n." );
319         }
320     }
321 }
322 }
323 }
324 /**
325 * Read "yes" or "no" from the Terminal.
326 * If an empty line or improper character is read,
327 * try again. No prompting is done.
328 * Look only at first character and accept any case.
329 *
330 * @return true if yes, false if no.
331 */
332 public boolean readYesOrNo()
333 {
334     while ( true ) {
335         char answer = readChar();
336         if ( answer == 'y' || answer == 'Y' ) {

```

```

337         return true;
338     }
339     else if ( answer == 'n' || answer == 'N' ) {
340         return false;
341     }
342 }
343 }
344 }
345 /**
346  * Read "yes" or "no" from the Terminal.
347  * If an empty line or improper character is read,
348  * throw an exception.
349  * Look only at first character and accept any case.
350  *
351  * @return true if yes, false if no.
352  *
353  * @throws RuntimeException on improper input.
354  */
355
356 public boolean readYesOrNoOnce()
357 {
358     char answer = readCharOnce( " (y or n): " );
359     if ( answer == 'y' || answer == 'Y' ) {
360         return true;
361     }
362     else if ( answer == 'n' || answer == 'N' ) {
363         return false;
364     }
365     else {
366         throw new RuntimeException( "Must be y or n." );
367     }
368 }
369
370 /**
371  * Read an integer, terminated by a new line, from the Terminal.
372  * If a NumberFormatException is encountered, try again.
373  *
374  * @param prompt output string to prompt for input.
375  * @return the input value as an int.
376  */
377
378 public int readInt( String prompt )
379 {
380     while( true ) {
381         try {
382             return Integer.
383                 parseInt(readNonNullLine( prompt ).trim());
384         }
385         catch (NumberFormatException e) {
386             println( "Not an integer. Please try again." );
387         }
388     }
389 }
390
391 /**
392  * Read an integer, terminated by a new line, from the Terminal.

```

```

393
394 *
395 * @param prompt output string to prompt for input.
396 * @return the input value as an int.
397 *
398 * @throws NumberFormatException for a badly formed integer.
399 */
400
401 public int readIntOnce( String prompt )
402 {
403     throws NumberFormatException
404     return Integer.parseInt(readNonNullLine( prompt ).trim());
405 }
406
407 /**
408  * Read an integer, terminated by a new line, from the Terminal.
409  * If a NumberFormatException is encountered, try again.
410  *
411  * @return the input value as an int.
412  */
413
414 public int readInt()
415 {
416     return readInt("");
417 }
418
419 /**
420  * Read an integer, terminated by a new line, from the Terminal.
421  *
422  * @return the input value as an int.
423  *
424  * @throws NumberFormatException for a badly formed integer.
425  */
426
427 public int readIntOnce()
428 {
429     throws NumberFormatException
430     return readIntOnce("");
431 }
432
433 /**
434  * Read a double-precision floating point number,
435  * terminated by a newline, from the Terminal.
436  * If a NumberFormatException is encountered, try again.
437  *
438  * @param prompt output string to prompt for input.
439  * @return the input value as a double.
440  */
441
442 public double readDouble( String prompt )
443 {
444     while( true ) {
445         try {
446             return Double.
447                 parseDouble(readNonNullLine( prompt ).trim());
448         }
449         catch (NumberFormatException e) {

```

```

449         println("Not a floating point number. Please try again.");
450     }
451 }
452 }
453 }
454 /**
455  * Read a double-precision floating point number,
456  * terminated by a newline, from the Terminal.
457  *
458  * @param prompt output string to prompt for input.
459  * @return the input value as a double.
460  *
461  * @throws NumberFormatException for a badly formed number.
462  */
463 public double readDoubleOnce( String prompt )
464     throws NumberFormatException
465 {
466     return Double.parseDouble(readNonNullLine( prompt ).trim());
467 }
468
469 /**
470  * Read a double-precision floating point number,
471  * terminated by a newline, from the Terminal.
472  *
473  * If a NumberFormatException is encountered, try again.
474  *
475  * @return the input value as a double.
476  */
477
478 public double readDouble()
479 {
480     return readDouble("");
481 }
482
483 /**
484  * Read a double-precision floating point number,
485  * terminated by a newline, from the Terminal.
486  *
487  * @return the input value as a double.
488  *
489  * @throws NumberFormatException for a badly formed number.
490  */
491
492 public double readDoubleOnce()
493     throws NumberFormatException
494 {
495     return readDouble("");
496 }
497
498 /**
499  * Print a Boolean value
500  * (<code>true</code> or <code>false</code>)
501  * to standard output (without a newline).
502  *
503  * @param b Boolean to print.
504  */

```

```

505 public void print( boolean b )
506 {
507     System.out.print( b );
508 }
509 }
510 }
511 /**
512  * Print character to standard output (without a newline).
513  *
514  * @param ch character to print.
515  *
516  * @throws NumberFormatException for a badly formed number.
517  */
518 public void print( char ch )
519 {
520     System.out.print( ch );
521 }
522
523 /**
524  * Print character array to standard output (without a newline).
525  *
526  * @param s character array to print.
527  */
528 public void print( char[] s )
529 {
530     System.out.print( s );
531 }
532
533 /**
534  * Print a double-precision floating point number to standard
535  * output (without a newline).
536  *
537  * @param val number to print.
538  */
539
540 public void print( double val )
541 {
542     System.out.print( val );
543 }
544
545 /**
546  * Print a floating point number to standard output
547  * (without a newline).
548  *
549  * @param val number to print.
550  */
551
552 public void print( float val )
553 {
554     System.out.print( val );
555 }
556
557 /**
558  * Print integer to standard output (without a newline).
559  *
560  * @param val integer to print.

```

```

561 */
562 public void print( int val )
563 {
564     System.out.println( val );
565 }
566
567 /**
568  * Print a long integer to standard output (without a newline).
569  *
570  * @param val integer to print.
571  */
572
573 public void print( long val )
574 {
575     System.out.println( val );
576 }
577
578 /**
579  * Print Object to standard output (without a newline).
580  *
581  * @param val Object to print.
582  */
583
584 public void print( Object val )
585 {
586     System.out.println( val.toString() );
587 }
588
589 /**
590  * Print string to standard output (without a newline).
591  *
592  * @param str String to print.
593  */
594
595 public void print( String str )
596 {
597     System.out.println( str );
598 }
599
600 /**
601  * Print a newline to standard output,
602  * terminating the current line.
603  */
604
605 public void println()
606 {
607     System.out.println();
608 }
609
610 /**
611  * Print a Boolean value
612  * (<code>true</code> or <code>false</code>)
613  * to standard output, followed by a newline.
614  * @param b Boolean to print.
615  */
616

```

```

617
618 public void println( boolean b )
619 {
620     System.out.println( b );
621 }
622
623 /**
624  * Print character to standard output, followed by a newline.
625  *
626  * @param ch character to print.
627  */
628
629 public void println( char ch )
630 {
631     System.out.println( ch );
632 }
633
634 /**
635  * Print a character array to standard output,
636  * followed by a newline.
637  *
638  * @param s character array to print.
639  */
640
641 public void println( char[] s )
642 {
643     System.out.println( s );
644 }
645
646 /**
647  * Print floating point number to standard output,
648  * followed by a newline.
649  *
650  * @param val number to print.
651  */
652
653 public void println( float val )
654 {
655     System.out.println( val );
656 }
657
658 /**
659  * Print a double-precision floating point number to standard
660  * output, followed by a newline.
661  *
662  * @param val number to print.
663  */
664
665 public void println( double val )
666 {
667     System.out.println( val );
668 }
669
670 /**
671  * Print integer to standard output, followed by a newline.
672  */

```



```

673      * @param val Integer to print.
674      */
675      public void println( int val )
676      {
677          System.out.println( val );
678      }
679      /**
680      * Print a long integer to standard output,
681      * followed by a newline.
682      * @param val Long integer to print.
683      */
684      public void println( long val )
685      {
686          System.out.println( val );
687      }
688      /**
689      * Print Object to standard output, followed by a newline.
690      * @param val Object to print
691      */
692      public void println( Object val )
693      {
694          System.out.println( val.toString() );
695      }
696      /**
697      * Print string to standard output, followed by a newline.
698      * @param str String to print
699      */
700      public void println( String str )
701      {
702          System.out.println( str );
703      }
704      /**
705      * Print string to standard output, followed by a newline.
706      * @param str String to print
707      */
708      public void println( String str )
709      {
710          System.out.println( str );
711      }
712      /**
713      * Print a Boolean value
714      * (<code>true</code> or <code>false</code>)
715      * to standard err (without a newline).
716      * @param b Boolean to print.
717      */
718      public void errPrint( boolean b )
719      {
720          System.err.print( b );
721      }
722      /**
723      * Print a Boolean value
724      * (<code>true</code> or <code>false</code>)
725      * to standard err (without a newline).
726      * @param b Boolean to print.
727      */
728      public void errPrint( boolean b )

```

```

729      * Print character to standard err (without a newline).
730      * @param ch character to print.
731      */
732      public void errPrint( char ch )
733      {
734          System.err.print( ch );
735      }
736      /**
737      * Print character array to standard err (without a newline).
738      * @param s character array to print.
739      */
740      public void errPrint( char[] s )
741      {
742          System.err.print( s );
743      }
744      /**
745      * Print a double-precision floating point number to standard
746      * err (without a newline).
747      * @param val number to print.
748      */
749      public void errPrint( double val )
750      {
751          System.err.print( val );
752      }
753      /**
754      * Print a floating point number to standard err
755      * (without a newline).
756      * @param val number to print.
757      */
758      public void errPrint( float val )
759      {
760          System.err.print( val );
761      }
762      /**
763      * Print a floating point number to standard err
764      * (without a newline).
765      * @param val number to print.
766      */
767      public void errPrint( float val )
768      {
769          System.err.print( val );
770      }
771      /**
772      * Print integer to standard err (without a newline).
773      * @param val integer to print.
774      */
775      public void errPrint( int val )
776      {
777          System.err.print( val );
778      }
779      /**
780      * Print integer to standard err (without a newline).
781      * @param val integer to print.
782      */
783      public void errPrint( int val )
784      {
785          System.err.print( val );
786      }

```

```

785      /**
786       * Print a long integer to standard err (without a newline).
787       *
788       * @param val integer to print.
789       */
790      public void errPrint( long val )
791      {
792          System.err.print( val );
793      }
794
795      /**
796       * Print Object to standard err (without a newline).
797       *
798       * @param val Object to print.
799       */
800      public void errPrint( Object val )
801      {
802          System.err.print( val.toString() );
803      }
804
805      /**
806       * Print string to standard err (without a newline).
807       *
808       * @param str String to print.
809       */
810      public void errPrint( String str )
811      {
812          System.err.print( str );
813      }
814
815      /**
816       * Print a newline to standard err,
817       * terminating the current line.
818       */
819      public void errPrintln()
820      {
821          System.err.println();
822      }
823
824      /**
825       * Print a Boolean value
826       * (<code>true</code> or <code>false</code>)
827       * to standard err, followed by a newline.
828       */
829      public void errPrintln( boolean b )
830      {
831          System.err.println( b );
832      }
833
834      /**
835       * Print integer to standard err, followed by a newline.
836       */
837      public void errPrintln( int val )
838      {
839          System.err.println( val );
840      }

```

```

841      /**
842       * Print character to standard err, followed by a newline.
843       *
844       * @param ch character to print.
845       */
846      public void errPrintln( char ch )
847      {
848          System.err.println( ch );
849      }
850
851      /**
852       * Print a character array to standard err,
853       * followed by a newline.
854       *
855       * @param s character array to print.
856       */
857      public void errPrintln( char[] s )
858      {
859          System.err.println( s );
860      }
861
862      /**
863       * Print floating point number to standard err,
864       * followed by a newline.
865       *
866       * @param val number to print.
867       */
868      public void errPrintln( float val )
869      {
870          System.err.println( val );
871      }
872
873      /**
874       * Print a double-precision floating point number to
875       * standard err, followed by a newline.
876       */
877      public void errPrintln( double val )
878      {
879          System.err.println( val );
880      }
881
882      /**
883       * Print integer to standard err, followed by a newline.
884       *
885       * @param val integer to print.
886       */
887      public void errPrintln( int val )
888      {
889          System.err.println( val );
890      }
891
892      /**
893       * Print integer to standard err, followed by a newline.
894       *
895       * @param val integer to print.
896       */
897      public void errPrintln( int val )
898      {
899          System.err.println( val );
900      }

```

```

897     }
898     /**
899     * Print a long integer to standard err, followed by a newline.
900     */
901     * @param val long integer to print.
902     */
903     public void errPrintln( long val )
904     {
905         System.err.println( val );
906     }
907     /**
908     * Print Object to standard err, followed by a newline.
909     */
910     * @param val Object to print
911     */
912     public void errPrintln( Object val )
913     {
914         System.err.println( val.toString() );
915     }
916     /**
917     * Print string to standard err, followed by a newline.
918     */
919     * @param str String to print
920     */
921     public void errPrintln( String str )
922     {
923         System.err.println( str );
924     }
925     /**
926     * Unit test for Terminal.
927     */
928     * @param args command line arguments:
929     * <pre>
930     * -e echo all input.
931     * </pre>
932     */
933     public static void main( String[] args )
934     {
935         Terminal t =
936             new Terminal( args.length == 1 && args[0].equals("-e") );
937         String line = t.readLine( "line:" );
938         String word = t.readWord( "word:" );
939         char c = t.readChar( "char:" );
940         boolean yn = t.readYesOrNo( "yorn:" );
941         double d = t.readDouble( "double:" );
942         int i = t.readInt( "int:" );
943
944
945
946
947
948
949
950
951
952

```

```

953         t.print( " line:[" ]; t.print( line ); t.print( "" );
954         t.print( " line:[" ]; t.println( line ); t.print( "" );
955         t.print( " word:[" ]; t.print( word ); t.print( "" );
956         t.print( " word:[" ]; t.println( word ); t.print( "" );
957         t.print( " char:[" ]; t.print( c ); t.print( "" );
958         t.print( " char:[" ]; t.println( c ); t.print( "" );
959         t.print( " yorn:[" ]; t.print( yn ); t.print( "" );
960         t.print( " yorn:[" ]; t.println( yn ); t.print( "" );
961         t.print( " doub:[" ]; t.print( d ); t.print( "" );
962         t.print( " doub:[" ]; t.println( d ); t.print( "" );
963         t.print( " int:[" ]; t.print( i ); t.print( "" );
964         t.print( " int:[" ]; t.println( i ); t.print( "" );
965         t.errPrint( " line:[" ]; t.errPrint( line ); t.errPrint( "" );
966         t.errPrint( " line:[" ]; t.errPrintln( line ); t.errPrint( "" );
967         t.errPrint( " word:[" ]; t.errPrint( word ); t.errPrint( "" );
968         t.errPrint( " word:[" ]; t.errPrintln( word ); t.errPrint( "" );
969         t.errPrint( " char:[" ]; t.errPrint( c ); t.errPrint( "" );
970         t.errPrint( " char:[" ]; t.errPrintln( c ); t.errPrint( "" );
971         t.errPrint( " yorn:[" ]; t.errPrint( yn ); t.errPrint( "" );
972         t.errPrint( " yorn:[" ]; t.errPrintln( yn ); t.errPrint( "" );
973         t.errPrint( " doub:[" ]; t.errPrint( d ); t.errPrint( "" );
974         t.errPrint( " doub:[" ]; t.errPrintln( d ); t.errPrint( "" );
975         t.errPrint( " int:[" ]; t.errPrint( i ); t.errPrint( "" );
976         t.errPrint( " int:[" ]; t.errPrintln( i ); t.errPrint( "" );
977
978
979
980
981
982
983
984
985
986
987
988
989     }

```

```

1 // foj/8/juno/Password.java//
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * Model a good password.
8  *
9  * <p>
10 * A password is a String satisfying the following conditions
11 * (close to those required of Unix passwords, according to
12 * the <code> man passwd </code> command in Unix):
13 * <br>
14 * <ul>
15 * <li> A password must have at least PASSWORD_LENGTH characters, where
16 * PASSWORD_LENGTH defaults to 6. Only the first eight characters
17 * are significant.
18 *
19 * <li> A password must contain at least two alphabetic characters
20 * and at least one numeric or special character. In this case,
21 * "alphabetic" refers to all upper or lower case letters.
22 *
23 * <li> A password must not contain a specified string as a substring
24 * For comparison purposes, an upper case letter and its
25 * corresponding lower case letter are equivalent.
26 *
27 * <li> A password must not be a substring of a specified string.
28 * For comparison purposes, an upper case letter and its
29 * corresponding lower case letter are equivalent.
30 *
31 * </ul>
32 * <br>
33 * A Password string may be stored in a Password object only in
34 * encrypted form.
35 */
36
37 public class Password
38 {
39     private String password;
40
41     /**
42      * Construct a new Password.
43      *
44      * @param password the new password.
45      * @param notSubstringOf a String that may not contain the password.
46      * @param doesNotContain a String the password may not contain.
47      *
48      * @exception BadPasswordException when password is unacceptable.
49      */
50
51     public Password(String password, String notSubstringOf,
52                     String doesNotContain)
53         throws BadPasswordException
54     {
55         // if password is not acceptable
56         // throw new BadPasswordException( reason )

```

```

57     }
58     }
59     // Rewrite s in a form that makes it hard to guess s.
60     private String encrypt( String s )
61     {
62         return Integer.toHexString(s.hashCode());
63     }
64 }
65
66 /**
67  * See whether a supplied guess matches this password.
68  *
69  * @param guess the trial password.
70  *
71  * @exception BadPasswordException when match fails.
72  */
73
74     public void match(String guess)
75     throws BadPasswordException
76     {
77     }
78
79     /**
80      * Unit test for Password objects.
81      */
82
83     public static void main( String[] args )
84     {
85     }
86
87 }

```

```
1 // foj/8/juno/BadPasswordException.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * The exception thrown when an initial password is unacceptable
8  * or a match against an existing password fails.
9  */
10
11 public class BadPasswordException extends Exception
12 {
13     BadPasswordException()
14     {
15         super();
16     }
17
18     BadPasswordException(String message)
19     {
20         super(message);
21     }
22 }
```

```

1 // fo1/9/copy/Copy1.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.io.*;
7
8 /**
9  * Simple read-a-char, write-a-char loop to exercise file I/O.
10  *
11  * Usage: java Copy1 inputFile outputFile
12  */
13
14 public class Copy1
15 {
16     private static final int EOF = -1; // end of file character rep.
17
18     /**
19      * All work is done here.
20      *
21      * @param args names of the input file and output file.
22      */
23
24     public static void main( String[] args )
25     {
26         FileReader inStream = null;
27         FileWriter outStream = null;
28         int ch;
29
30         try {
31             // open the files
32             inStream = new FileReader( args[0] );
33             outStream = new FileWriter( args[1] );
34
35             // copy
36             while ((ch = inStream.read()) != EOF) {
37                 outStream.write( ch );
38             }
39         }
40         catch (IndexOutOfBoundsException e) {
41             System.err.println(
42                 "usage: java Copy1 sourcefile targetfile" );
43         }
44         catch (FileNotFoundException e) {
45             System.err.println( e ); // rely on e's toString()
46         }
47         catch (IOException e) {
48             System.err.println( e );
49         }
50         finally { // close the files
51             try {
52                 if (inStream != null) {
53                     inStream.close();
54                 }
55             }
56             catch (Exception e) {

```

```

57         }
58     }
59     try {
60         if (outStream != null) {
61             outStream.close();
62         }
63     }
64     catch (Exception e) {
65         System.err.println("Unable to close output stream.");
66     }
67 }
68 }
69 }

```

```

1 // fo1/9/copy/Copy2.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.io.*;
7
8 /**
9  * Simple read-a-line write-a-line loop to exercise file I/O.
10  *
11  * Usage: java Copy2 inputFile outputFile
12  */
13
14 public class Copy2
15 {
16     /**
17      * All work is done here.
18      *
19      * @param args names of the input file and output file.
20      */
21
22     public static void main( String[] args )
23     {
24         BufferedReader inStream = null;
25         BufferedWriter outStream = null;
26         String line;
27
28         try {
29             // open the files
30             inStream = new BufferedReader(new FileReader(args[0]));
31             outStream = new BufferedWriter(new FileWriter(args[1]));
32
33             // copy
34             while ((line = inStream.readLine()) != null) {
35                 outStream.write( line );
36                 outStream.newLine();
37             }
38         }
39         catch (IndexOutOfBoundsException e) {
40             System.err.println(
41                 "usage: java Copy2 sourcefile targetfile" );
42         }
43         catch (FileNotFoundException e) {
44             System.err.println( e ); // rely on e's toString()
45         }
46         catch (IOException e) {
47             System.err.println( e );
48         }
49         finally { // close the files
50             try {
51                 if (inStream != null) {
52                     inStream.close();
53                 }
54             }
55             catch (Exception e) {
56                 System.err.println("Unable to close input stream.");

```

```

57     }
58     try {
59         if (outStream != null) {
60             outStream.close();
61         }
62     }
63     catch (Exception e) {
64         System.err.println("Unable to close output stream.");
65     }
66 }
67 }
68 }

```

```

1 // fo1/9/bank/Bank.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7 import java.io.*;
8
9 /**
10  * A Bank object simulates the behavior of a simple bank/ATM.
11  * It contains a Terminal object and a collection of
12  * BankAccount objects.
13
14  * The visit method opens this Bank for business,
15  * prompting the customer for input.
16
17  * It is persistent: it can save its state to a file and read it
18  * back at a later time.
19
20  * To create a Bank and open it for business issue the command
21  * <code>java Bank</code> with appropriate arguments.
22
23  * @see BankAccount
24  * @version 9
25  */
26
27 public class Bank
28     implements Serializable
29 {
30     private String bankName; // the name of this Bank
31     private transient Terminal atm; // for communication with world
32     private int balance = 0; // total cash on hand
33     private int transactionCount = 0; // number of Bank transactions
34     private Month month; // the current month.
35     private Map accountList; // mapping names to accounts.
36
37     private int checkFee = 2; // cost for each check
38     private int transactionFee = 1; // fee for each transaction
39     private int monthlyCharge = 5; // monthly charge
40     private double interestRate = 0.05; // annual rate paid on savings
41     private int maxFreeTransactions = 3; // for savings accounts
42
43     // what the banker can ask of the bank
44
45     private static final String BANKER_COMMANDS =
46         "Banker commands: " +
47         "exit, open, customer, nextmonth, report, help.";
48
49     // what the customer can ask of the bank
50
51     private static final String CUSTOMER_TRANSACTIONS =
52         " Customer transactions: deposit, withdraw, transfer,\n" +
53         " balance, cash check, quit, help.";
54
55     /**
56      * Construct a Bank with the given name.

```

```

57
58     * @param bankName the name for this Bank.
59     */
60
61     public Bank( String bankName )
62     {
63         this.atm = atm;
64         this.bankName = bankName;
65         accountList = new TreeMap();
66         month = new Month();
67     }
68
69     /**
70      * Simulates interaction with a Bank.
71      * Presents the user with an interactive loop, prompting for
72      * banker transactions and in the case of the banker
73      * transaction "customer", an account id and further
74      * customer transactions.
75      */
76
77     public void visit()
78     {
79         instructUser();
80
81         String command;
82         while ( ! (command =
83             atm.readWord("banker command: ").equals("exit")) ) {
84
85             if (command.startsWith("h")) {
86                 help( BANKER_COMMANDS );
87             }
88             else if (command.startsWith("o")) {
89                 openNewAccount();
90             }
91             else if (command.startsWith("n")) {
92                 newMonth();
93             }
94             else if (command.startsWith("r")) {
95                 report();
96             }
97             else if (command.startsWith("c") ) {
98                 BankAccount acct = whichAccount();
99                 if ( acct != null ) {
100                     processTransactionsForAccount( acct );
101                 }
102             }
103             else {
104                 // Unrecognized Request
105                 atm.println( "unknown command: " + command );
106             }
107         }
108         report();
109         atm.println( "Goodbye from " + bankName );
110     }
111 }
112

```



```

113 // Open a new bank account,
114 // prompting the user for information.
115
116 private void openNewAccount()
117 {
118     String accountName = atm.readWord( "Account name: " );
119     char accountType =
120     atm.readChar( "Type of account (r/c/f/s): " );
121     try {
122         int startup = readPosAmt( "Initial deposit: " );
123         BankAccount newAccount;
124         switch( accountType ) {
125             case 'c':
126                 newAccount = new CheckingAccount( startup, this );
127                 break;
128             case 'f':
129                 newAccount = new FeeAccount( startup, this );
130                 break;
131             case 's':
132                 newAccount = new SavingsAccount( startup, this );
133                 break;
134             case 'r':
135                 newAccount = new RegularAccount( startup, this );
136                 break;
137             default:
138                 atm.println( "invalid account type: " + accountType );
139                 return;
140         }
141         accountList.put( accountName, newAccount );
142         atm.println( "opened new account " + accountName
143                     + " with $" + startup );
144     }
145     catch (NegativeAmountException e) {
146         atm.errPrintln(
147             "You cannot open an account with a negative balance");
148     }
149     catch (InsufficientFundsException e) {
150         atm.errPrintln( "Initial deposit doesn't cover fee" );
151     }
152 }
153
154 // Prompt the customer for transaction to process.
155 // Then send an appropriate message to the account.
156
157 private void processTransactionsForAccount( BankAccount acct )
158 {
159     help( CUSTOMER_TRANSACTIONS );
160
161     String transaction;
162     while ( !(transaction =
163             atm.readWord( " transaction: ")).equals("quit")) {
164
165         try {
166             if ( transaction.startsWith( "h" ) ) {
167                 help( CUSTOMER_TRANSACTIONS );
168

```

```

169         else if ( transaction.startsWith( "d" ) ) {
170             int amount = readPosAmt( " amount: " );
171             atm.println( " deposited "
172                         + acct.deposit( amount ) );
173         }
174         else if ( transaction.startsWith( "w" ) ) {
175             int amount = readPosAmt( " amount: " );
176             atm.println( " withdrew "
177                         + acct.withdraw( amount ) );
178         }
179         else if ( transaction.startsWith( "c" ) ) {
180             int amount = readPosAmt( " amount of check: " );
181             try { // to cast acct to CheckingAccount ...
182                 atm.println( " cashed check for " +
183                             ((CheckingAccount) acct).honorCheck( amount ) )
184             }
185             catch (ClassCastException e) {
186                 // if not a checking account, report error
187                 atm.errPrintln(
188                     " Sorry, not a checking account. " );
189             }
190         }
191         else if ( transaction.startsWith( "t" ) ) {
192             atm.print( " to " );
193             BankAccount toacct = whichAccount();
194             if ( toacct != null ) {
195                 int amount = readPosAmt( " amount to transfer: " );
196                 atm.println( " transferred "
197                             + toacct.deposit( acct.withdraw( amount ) ) );
198             }
199         }
200         else if ( transaction.startsWith( "b" ) ) {
201             atm.println( " current balance "
202                         + acct.requestBalance() );
203         }
204         else {
205             atm.println( " sorry, unknown transaction " );
206         }
207     }
208     catch (InsufficientFundsException e) {
209         atm.errPrintln( " Insufficient funds " +
210                         e.getMessage() );
211     }
212     catch (NegativeAmountException e) {
213         atm.errPrintln( " Sorry, negative amounts disallowed. " );
214     }
215     atm.println();
216 }
217
218 // Prompt for an account name (or number), look it up
219 // in the account list. If it's there, return it;
220 // otherwise report an error and return null.
221
222 private BankAccount whichAccount()
223 {
224

```

```

225     String accountName = atm.readWord( "account name: " );
226     BankAccount account = (BankAccount) accountList.get(accountName);
227     if (account == null) {
228         atm.println( "not a valid account" );
229     }
230     return account;
231 }
232
233 // Action to take when a new month starts.
234 // Update the month field by sending a next message.
235 // Loop on all accounts, sending each a newMonth message.
236
237 private void newMonth()
238 {
239     month.next();
240     Iterator i = accountList.keySet().iterator();
241     while ( i.hasNext() ) {
242         String name = (String) i.next();
243         BankAccount acct = (BankAccount) accountList.get( name );
244         try {
245             acct.newMonth();
246         }
247         catch ( InsufficientFundsException ) {
248             atm.errPrintln( "Insufficient funds in account \"\" +
249                 name + "\" for monthly fee" );
250         }
251     }
252 }
253
254 // Report bank activity.
255 // For each BankAccount, print the customer id (name or number),
256 // account balance and the number of transactions.
257 // Then print Bank totals.
258
259 private void report()
260 {
261     atm.println( bankName + " report for " + month );
262     atm.println( "\nSummaries of individual accounts:" );
263     atm.println( "account balance transaction count" );
264     for ( Iterator i = accountList.keySet().iterator();
265         i.hasNext(); ) {
266         String accountName = (String) i.next();
267         BankAccount acct = (BankAccount) accountList.get(accountName)
268             atm.println(accountName + "\t$" + acct.getBalance() + "\t\t"
269                 + acct.getTransactionCount());
270     }
271     atm.println( "\nBank totals" );
272     atm.println( "open accounts: " + getNumberOfAccounts() );
273     atm.println( "cash on hand: $" + getBalance() );
274     atm.println( "transactions: " + getTransactionCount() );
275     atm.println();
276 }
277
278 // Welcome the user to the bank and instruct her on
279 // her options.
280

```

```

281
282     private void instructUser()
283     {
284         atm.println( "Welcome to " + bankName );
285         atm.println( month.toString() );
286         atm.println( "Open some accounts and work with them." );
287         help( BANKER_COMMANDS );
288     }
289
290 // Display a help string.
291
292 private void help( String helpString )
293 {
294     atm.println( helpString );
295     atm.println();
296 }
297
298 // Read amount prompted for from the atm.
299 // Throw a NegativeAmountException if amount < 0
300
301 private int readPosAmt( String prompt )
302     throws NegativeAmountException
303 {
304     int amount = atm.readInt( prompt );
305     if (amount < 0) {
306         throw new NegativeAmountException();
307     }
308     return amount;
309 }
310
311 /**
312  * Increment bank balance by given amount.
313  */
314 * @param amount the amount increment.
315 */
316
317 public void incrementBalance(int amount)
318 {
319     balance += amount;
320 }
321
322 /**
323  * Increment by one the count of transactions,
324  * for this bank.
325  */
326
327 public void countTransaction()
328 {
329     transactionCount++;
330 }
331
332 /**
333  * Get the number of transactions performed by this bank.
334  */
335 * @return number of transactions performed.
336 */

```

```

337 public int getTransactionCount( )
338 {
339     return transactionCount ;
340 }
341 }
342 /**
343  * The charge this bank levies for cashing a check.
344  * @return check fee
345  */
346 * @return check fee
347 */
348
349 public int getCheckFee( )
350 {
351     return checkFee ;
352 }
353
354 /**
355  * The charge this bank levies for a transaction.
356  * @return the transaction fee
357  */
358 */
359
360 public int getTransactionFee( )
361 {
362     return transactionFee ;
363 }
364
365 /**
366  * The charge this bank levies each month.
367  * @return the monthly charge
368  */
369 */
370
371 public int getMonthlyCharge( )
372 {
373     return monthlyCharge;
374 }
375
376 /**
377  * The current interest rate on savings.
378  * @return the interest rate
379  */
380 */
381
382 public double getInterestRate( )
383 {
384     return interestRate;
385 }
386
387 /**
388  * The number of free transactions per month.
389  * @return the number of transactions
390  */
391 */
392

```

```

393 public int getMaxFreeTransactions( )
394 {
395     return maxFreeTransactions;
396 }
397 }
398 /**
399  * Get the current bank balance.
400  * @return current bank balance.
401  */
402 */
403
404 public int getBalance( )
405 {
406     return balance;
407 }
408
409 /**
410  * Get the current number of open accounts.
411  * @return number of open accounts.
412  */
413 */
414
415 public int getNumberOfAccounts( )
416 {
417     return accountList.size();
418 }
419
420 /**
421  * Set the atm for this Bank.
422  * @param atm the Bank's atm.
423  */
424 */
425
426 public void setAtm( Terminal atm ) {
427     this.atm = atm;
428 }
429
430 /**
431  * Run the simulation by creating and then visiting a new Bank.
432  * <p>
433  * A -e argument causes the input to be echoed.
434  * This can be useful for executing the program against
435  * a test script, e.g.,
436  * <pre>
437  * java Bank -e < Bank.in
438  * </pre>
439  *
440  * The -f argument reads the bank's state from the specified
441  * file, and writes it to that file when the program exits.
442  *
443  * @param args the command line arguments:
444  * <pre>
445  * -e echo input.
446  * -f filename
447  * bankName any other command line argument.
448

```

```

449  *      </pre>
450  */
451
452  public static void main( String[] args )
453  {
454      boolean echo      = false;
455      String bankName   = null;
456      String bankName   = "Persistent Bank";
457      Bank theBank      = null;
458
459      // parse the command line arguments
460      for (int i = 0; i < args.length; i++ ) {
461          if (args[i].equals("-e")) { // echo input to output
462              echo = true;
463              continue;
464          }
465          if (args[i].equals("-f")) { // read/write Bank from/to file
466              bankName = args[i+1];
467              continue;
468          }
469      }
470
471      // create a new Bank or read one from a file
472      if (bankName == null) {
473          theBank = new Bank( bankName );
474      }
475      else {
476          theBank = readBank( bankName, bankName );
477      }
478
479      // give the Bank a Terminal, then visit
480      theBank.setAtm(new Terminal(echo));
481      theBank.visit();
482
483      // write theBank's state to a file if required
484      if (bankName != null) {
485          writeBank(theBank, bankName);
486      }
487  }
488
489  // Read a Bank from a file (create it if file doesn't exist).
490  //
491  // @param bankName   the name of the Bank
492  // @param bankName the name of the file containing the Bank
493  //
494  // @return the Bank
495
496  private static Bank readBank(String bankName, String bankName)
497  {
498      File file = new File( bankName );
499      if (!file.exists()) {
500          return new Bank( bankName );
501      }
502      ObjectInputStream inStream = null;
503      try {
504          inStream = new ObjectInputStream(

```

```

505          new FileInputStream( file ) );
506      Bank bank = (Bank) inStream.readObject();
507      System.out.println(
508          "Bank state read from file " + bankName);
509      return bank;
510  }
511  catch (Exception e ) {
512      System.err.println(
513          "Problem reading " + bankName );
514      System.err.println(e);
515      System.exit(1);
516  }
517  finally {
518      try {
519          inStream.close();
520      }
521      catch (Exception e) {
522      }
523  }
524  return null; // you can never get here
525  }
526
527  // Write a Bank to a file.
528
529  // @param bank       the Bank
530  // @param fileName the name of the file to write the Bank to
531
532  private static void writeBank( Bank bank, String fileName)
533  {
534      ObjectOutputStream outStream = null;
535      try {
536          outStream = new ObjectOutputStream(
537              new FileOutputStream( fileName ) );
538          outStream.writeObject( bank );
539          System.out.println(
540              "Bank state written to file " + fileName);
541      }
542      catch (Exception e ) {
543          System.err.println(
544              "Problem writing " + fileName );
545      }
546      finally {
547          try {
548              outStream.close();
549          }
550          catch (Exception e ) {
551          }
552      }
553  }
554  }
555  }

```

```

1 // fo1/9/bank/BankAccount.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.io.Serializable;
7
8 /**
9  * A BankAccount object has private fields to keep track
10 * of its current balance, the number of transactions
11 * performed and the Bank in which it is an account, and
12 * and public methods to access those fields appropriately.
13  *
14  * @see Bank
15  * @version 9
16  */
17
18 public abstract class BankAccount
19 implements Serializable
20 {
21     private int balance = 0; // Account balance (whole dollars)
22     private int transactionCount = 0; // Number of transactions performed
23     private Bank issuingBank; // Bank issuing this account
24
25     /**
26      * Construct a BankAccount with the given initial balance and
27      * issuing Bank. Construction counts as this BankAccount's
28      * first transaction.
29      *
30      * @param initialBalance the opening balance.
31      * @param issuingBank the bank that issued this account.
32      *
33      * @exception InsufficientFundsException when appropriate.
34      */
35     protected BankAccount( int initialBalance, Bank issuingBank )
36     throws InsufficientFundsException
37     {
38         this.issuingBank = issuingBank;
39         deposit( initialBalance );
40     }
41
42     /**
43      * Get transaction fee. By default, 0.
44      *
45      * Override this for accounts having transaction fees.
46      *
47      * @return the fee.
48      */
49     protected int getTransactionFee()
50     {
51         return 0;
52     }
53 }
54
55 /**
56  * The bank that issued this account.

```

```

57  *
58  * @return the Bank.
59  */
60
61     protected Bank getIssuingBank()
62     {
63         return issuingBank;
64     }
65
66     /**
67      * Withdraw the given amount, decreasing this BankAccount's
68      * balance and the issuing Bank's balance.
69      *
70      * Counts as a transaction.
71      *
72      * @param amount the amount to be withdrawn
73      * @return amount withdrawn
74      *
75      * @exception InsufficientFundsException when appropriate.
76      */
77     public int withdraw( int amount )
78     throws InsufficientFundsException
79     {
80         incrementBalance( -amount - getTransactionFee() );
81         countTransaction();
82         return amount ;
83     }
84
85     /**
86      * Deposit the given amount, increasing this BankAccount's
87      * balance and the issuing Bank's balance.
88      *
89      * Counts as a transaction.
90      *
91      * @param amount the amount to be deposited
92      * @return amount deposited
93      *
94      * @exception InsufficientFundsException when appropriate.
95      */
96     public int deposit( int amount )
97     throws InsufficientFundsException
98     {
99         incrementBalance( amount - getTransactionFee() );
100         countTransaction();
101         return amount ;
102     }
103
104     /**
105      * Request for balance. Counts as a transaction.
106      *
107      * @return current account balance.
108      *
109      * @exception InsufficientFundsException when appropriate.
110      */
111     public int requestBalance()
112

```

```

113     throws InsufficientFundsException
114     {
115         incrementBalance( - getTransactionFee() );
116         countTransaction();
117         return getBalance() ;
118     }
119 }
120
121 /**
122  * Get the current balance.
123  * Does NOT count as a transaction.
124  */
125 @return current account balance
126 */
127
128 public int getBalance()
129 {
130     return balance;
131 }
132
133 /**
134  * Increment account balance by given amount.
135  * Also increment issuing Bank's balance.
136  * Does NOT count as a transaction.
137  */
138 @param amount the amount of the increment.
139 * @exception InsufficientFundsException when appropriate.
140 */
141
142 public final void incrementBalance( int amount )
143     throws InsufficientFundsException
144     {
145         int newBalance = balance + amount;
146         if (newBalance < 0) {
147             throw new InsufficientFundsException(
148                 "For this transaction" );
149         }
150         balance = newBalance;
151         getIssuingBank().incrementBalance( amount );
152     }
153 }
154
155 /**
156  * Get the number of transactions performed by this
157  * account. Does NOT count as a transaction.
158  */
159 @return number of transactions performed.
160 */
161
162 public int getTransactionCount()
163 {
164     return transactionCount;
165 }
166
167 /**
168  * Increment by 1 the count of transactions, for this account
169  * and for the issuing Bank.

```

```

169     * Does NOT count as a transaction.
170     *
171     * @exception InsufficientFundsException when appropriate.
172     */
173 }
174
175 public void countTransaction()
176     throws InsufficientFundsException
177     {
178         transactionCount++;
179         this.getIssuingBank().countTransaction();
180     }
181 }
182
183 /**
184  * Action to take when a new month starts.
185  *
186  * @exception InsufficientFundsException thrown when funds
187  * on hand are not enough to cover the fees.
188  */
189
190 public abstract void newMonth()
191     throws InsufficientFundsException;

```

```

1 // foj/9/bank/class Month
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.io.*;
7 import java.util.Calendar;
8
9 /**
10  * The Month class implements an object that keeps
11  * track of the month of the year.
12  *
13  * @version 9
14  */
15
16 public class Month
17     implements Serializable
18 {
19     private static final String[] monthName =
20         { "Jan", "Feb", "Mar", "Apr", "May", "Jun",
21           "Jul", "Aug", "Sep", "Oct", "Nov", "Dec" };
22
23     private int month;
24     private int year;
25
26     /**
27      * Month constructor constructs a Month object
28      * initialized to the current month and year.
29      */
30
31     public Month()
32     {
33         Calendar rightNow = Calendar.getInstance();
34         month = rightNow.get( Calendar.MONTH );
35         year = rightNow.get( Calendar.YEAR );
36     }
37
38     /**
39      * Advance to next month.
40      */
41
42     public void next()
43     {
44         month = (month + 1) % 12;
45         if (month == 0) {
46             year++;
47         }
48     }
49
50     /**
51      * How a Month is displayed as a String -
52      * for example, "Jan, 2003".
53      *
54      * @return String representation of the month.
55      */
56

```

```

57     public String toString()
58     {
59         return monthName[month] + ", " + year;
60     }
61
62     /**
63      * For unit testing.
64      */
65
66     public static void main( String[] args )
67     {
68         Month m = new Month();
69         for (int i=0; i < 14; i++, m.next()) {
70             System.out.println(m);
71         }
72         for (int i=0; i < 35; i++, m.next()); // no loop body
73         System.out.println( "three years later: " + m );
74         for (int i=0; i < 120; i++, m.next()); // no loop body
75         System.out.println( "ten years later: " + m );
76     }
77 }

```

```

1 // joi/10/joi/JOIPanel.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.applet.*;
7 import java.awt.*;
8 import java.awt.event.*;
9
10 /**
11  * A JOIPanel displays a button and a message.
12  * Pushing the button changes the message.
13  *
14  * This panel can be displayed either from an applet
15  * in a browser or by the JVM as an application.
16  *
17  * @version 1.0
18  */
19
20 public class JOIPanel extends Applet
21 {
22     private static final String MESSAGE1 = "Java Outside In";
23     private static final String MESSAGE2 = "Java Inside Out";
24     private String currentMessage = MESSAGE1; // currently displayed
25
26     private Font font; // for printing the message
27     private Button button; // for changing messages
28
29     /**
30      * Equip this Panel with a Button
31      * and an associated ButtonListener, and
32      * set the font for the message.
33      */
34
35     public void init()
36     {
37         // what this Panel looks like
38         button = new Button( "Press Me" );
39         this.add( button );
40         font = new Font("Garamond", Font.BOLD, 48);
41
42         // how this Panel behaves
43         button.addActionListener( new JOIButtonListener( this ) );
44     }
45
46     /**
47      * Method that responds when the ButtonListener sends a
48      * changeMessage message.
49      */
50
51     public void changeMessage()
52     {
53         currentMessage =
54             currentMessage.equals(MESSAGE1) ? MESSAGE2 : MESSAGE1;
55         this.repaint();
56     }

```

```

57
58     /**
59      * Draw the current message on this Panel.
60      *
61      * (The button is already there.)
62      *
63      * @param g an object encapsulating the graphics (e.g. pen)
64      * properties.
65      */
66
67     public void paint(Graphics g)
68     {
69         g.setColor(Color.black);
70         g.setFont(font);
71         g.drawString(currentMessage, 40, 75);
72     }
73
74     /**
75      * Ask the JVM to display this Panel.
76      */
77
78     public static void main( String[] args )
79     {
80         Terminal t = new Terminal();
81         JFrame frame = new JFrame();
82         JOIPanel panel = new JOIPanel();
83         panel.init();
84         frame.add(panel);
85         frame.setSize(400, 120);
86         frame.show();
87         t.readLine("Type return to close the window ... ");
88         System.exit(0);
89     }
90 }

```



```
1 // joi/10/joi/ButtonListener.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.awt.event.*;
7
8 /**
9  * A simple listener for responding to button presses.
10  * It knows the Panel on which the button lives, and
11  * responds to button events by sending a changeMessage()
12  * to that Panel.
13  *
14  * @version 10
15  */
16
17 public class JOIButtonListener implements ActionListener
18 {
19     private JOIPanel panel; // the Panel containing the Button
20
21     /**
22      * Construct the ButtonListener.
23      *
24      * @param panel the Panel on which this Button will act.
25      */
26
27     public JOIButtonListener( JOIPanel panel )
28     {
29         this.panel = panel;
30     }
31
32     /**
33      * Defines the ActionListener behavior that must be implemented.
34      *
35      * When a user pushes the Button that we're listening to,
36      * send a changeMessage() message to the Panel.
37      *
38      * @param e the "event" when the button is pressed.
39      */
40
41     public void actionPerformed( ActionEvent e )
42     {
43         panel.changeMessage();
44     }
45 }
```

```
1 <!-- joi/10/joi.html-->
2 <!-- -->
3 <!-- -->
4 <!-- Copyright 2002 Bill Campbell and Ethan Bolker-->
5
6 <html>
7 <body>
8
9 <applet
10 code="JOIPanel.class" height=100 width=400>
11 </applet>
12
13 </html>
14 </body>
```

```

1 // joi/10/joiapplet/JOIApplet.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.applet.*;
7 import java.awt.*;
8 import java.awt.event.*;
9
10 /**
11  * A JOIApplet displays a button and a message.
12  * Pushing the button changes the message.
13  *
14  * This class provides both the panel and the listener for
15  * the button on the panel - a common GUI programming idiom.
16  *
17  * The panel can be displayed either from an applet
18  * in a browser or by the JVM as an application.
19  *
20  * @version 1.0
21  */
22
23
24 public class JOIApplet extends Applet implements ActionListener
25 {
26     private static final String MESSAGE1 = "Java Outside In";
27     private static final String MESSAGE2 = "Java Inside Out";
28     private String currentMessage = MESSAGE1; // currently displayed
29
30     private Font font; // for printing the message
31     private Button button; // for changing messages
32
33     /**
34      * Equip this Panel with a Button
35      * and an associated ActionListener, and
36      * set the font for the message.
37      */
38
39     public void init()
40     {
41         // what this Panel looks like
42         button = new Button( "Press Me" );
43         this.add( button );
44         font = new Font( "Garamond", Font.BOLD, 48 );
45
46         // how this Panel behaves
47         button.addActionListener( this );
48     }
49
50     /**
51      * Defines the ActionListener behavior that must be
52      * implemented.
53      *
54      * When a user pushes the Button that we're listening to,
55      * send a changeMessage() message to the Panel.
56

```

```

57     * @param e the "event" when the button is pressed.
58     */
59
60     public void actionPerformed( ActionEvent e )
61     {
62         currentMessage =
63             currentMessage.equals( MESSAGE1 ) ? MESSAGE2 : MESSAGE1;
64         this.repaint();
65     }
66
67     /**
68      * Draw the current message on this Panel.
69      *
70      * (The button is already there.)
71      *
72      * @param g an object encapsulating the graphics (e.g. pen)
73      * properties.
74      */
75
76     public void paint( Graphics g )
77     {
78         g.setColor( Color.black );
79         g.setFont( font );
80         g.drawString( currentMessage, 40, 75 );
81     }
82
83     /**
84      * Ask the JVM to display this Panel.
85      */
86
87     public static void main( String[] args )
88     {
89         Terminal t = new Terminal();
90         Frame frame = new Frame();
91         JOIApplet panel = new JOIApplet();
92         panel.init();
93         frame.add( panel );
94         frame.setSize( 400, 120 );
95         frame.show();
96         t.readLine( "Type return to close the window ... " );
97         System.exit( 0 );
98     }
99

```

```
1 <!-- joi/10/joiapplet/classes/joiapplet.html-->
2 <!-- -->
3 <!-- -->
4 <!-- Copyright 2002 Bill Campbell and Ethan Bolker-->
5
6 <html>
7 <body>
8
9 <applet
10 code="JoiApplet.class" height=100 width=400>
11 </applet>
12
13 </html>
14 </body>
```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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



```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```
1 // fo1/10/juno/MkdirCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * The Juno shell command to create a new directory.
10  * Usage:
11  * <pre>
12  *   mkdir directory-name
13  * </pre>
14  *
15  * @version 1.0
16  */
17
18 public class MkdirCommand extends ShellCommand
19 {
20     MkdirCommand()
21     {
22         super( "create a subdirectory of the current directory",
23              "directory-name" );
24     }
25
26     /**
27      * Create a new Directory in the current Directory.
28      *
29      * @param args the remainder of the command line.
30      * @param sh the current shell.
31      *
32      * @exception JunoException for reporting errors.
33      */
34
35     public void doit( StringTokenizer args, Shell sh )
36     {
37         throws JunoException
38     {
39         String filename = args.nextToken();
40         new Directory( filename, sh.getUser(), sh.getDot() );
41     }
42 }
```

```

1 // fo1/10/juno/TypeCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * The Juno shell command to display the contents of a
10 * text file.
11 * Usage:
12 * <pre>
13 *   type textfile
14 * </pre>
15 *
16 * @version 1.0
17 */
18
19 public class TypeCommand extends ShellCommand
20 {
21     TypeCommand()
22     {
23         super( "display contents of a TextFile", "textfile" );
24     }
25
26     /**
27     * Display the contents of a TextFile.
28     *
29     * @param args the remainder of the command line.
30     * @param sh the current Shell
31     *
32     * @exception JunoException for reporting errors
33     */
34
35     public void doIt( StringTokenizer args, Shell sh )
36     throws JunoException
37     {
38         String filename;
39
40         try {
41             filename = args.nextToken();
42         }
43         catch (NoSuchElementException e) {
44             throw new BadShellCommandException( this );
45         }
46         try {
47             sh.getConsole().println(
48                 ( (TextFile) sh.getDot() ).
49                 retrieveFile( filename ) ).getContents();
50         }
51         catch (NullPointerException e) {
52             throw new JunoException( "JFile does not exist: "
53                 + filename);
54         }
55         catch (ClassCastException e) {
56             throw new JunoException( "JFile not a text file: "

```

```

57     }
58 }
59 }

```



```
1 // fo1/10/juno/HelpCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * The Juno shell command to display help on the shell commands.
10  * Usage:
11  * <pre>
12  *     help
13  * </pre>
14  *
15  * @version 1.0
16  */
17
18 public class HelpCommand extends ShellCommand
19 {
20     HelpCommand()
21     {
22         super( "display ShellCommands" );
23     }
24
25     /**
26      * Print out help for all commands.
27      *
28      * @param args the remainder of the command line.
29      * @param sh the current shell
30      *
31      * @exception JunoException for reporting errors
32      */
33
34     public void doIt( StringTokenizer args, Shell sh )
35     {
36         throws JunoException
37     {
38         // Get command keys from global table, print them out.
39
40         sh.getConsole().println( "shell commands" );
41         ShellCommandTable table = sh.getSystem().getCommandTable();
42         String[] names = table.getCommandNames();
43         for (int i = 0; i < names.length; i++) {
44             String cmdname = names[i];
45             ShellCommand cmd =
46                 (ShellCommand) table.lookup( cmdname );
47             sh.getConsole().
48                 println( " " + cmdname + " : " + cmd.getHelpString() );
49         }
50     }
51 }
```



```
1 // fo1/10/juno/CdCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * The Juno shell command to change directory.
10  * Usage:
11  * <pre>
12  *   cd [directory]
13  * </pre>
14  * For moving to the named directory.
15  *
16  * @version 1.0
17  */
18
19 class CdCommand extends ShellCommand
20 {
21     CdCommand()
22     {
23         super( "change current directory", "[ directory ]" );
24     }
25
26     /**
27     * Move to the named directory
28     *
29     * @param args the remainder of the command line.
30     * @param sh the current shell
31     *
32     * @exception JunoException for reporting errors
33     */
34
35     public void doIt( StringTokenizer args, Shell sh )
36     throws JunoException
37     {
38         String dirname = "";
39         Directory d = sh.getUser().getHome(); // default
40         if ( args.hasMoreTokens() ) {
41             dirname = args.nextToken();
42             if (dirname.equals(".")) {
43                 if (sh.getDot().isRoot()) {
44                     d = sh.getDot(); // no change
45                 }
46                 else
47                     d = sh.getDot().getParent();
48             }
49             else if (dirname.equals("..")) {
50                 d = sh.getDot(); // no change
51             }
52             else {
53                 d = (Directory)(sh.getDot().retrieveFile(dirname));
54             }
55         }
56         sh.setDot( d );
57     }
58 }
```

57 }

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

```

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

```

```

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

```

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

```
1 // foj/10/juno/LogoutCommand.java
2 //
3 //
4 // Copyright 2003, Bill Campbell and Ethan Bolker
5
6 import java.util.*;
7
8 /**
9  * The Juno shell command to log out.
10  * Usage:
11  * <pre>
12  *     logout
13  * </pre>
14  *
15  * @version 1.0
16  */
17
18 public class LogoutCommand extends ShellCommand
19 {
20     LogoutCommand()
21     {
22         super( "log out, return to login: prompt" );
23     }
24
25     /**
26      * Log out from the current shell.
27      *
28      * @param args the remainder of the command line.
29      * @param sh the current shell
30      *
31      * @exception JunoException for reporting errors
32      */
33
34     public void doIt( StringTokenizer args, Shell sh )
35     {
36         throws JunoException
37     {
38         throw new ExitShellException();
39     }
39 }
```

```

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

```

```

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

```



```

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

```

```

169     }
170 }
171 /**
172  * A JFile whose parent is null is defined to be the root
173  * (of a tree).
174  */
175 * @return true when this JFile is the root.
176 */
177
178 public boolean isRoot()
179 {
180     return (parent == null);
181 }
182
183 /**
184  * How a JFile represents itself as a String.
185  * That is,
186  * <pre>
187  *   owner      size      modDate      name+suffix
188  * </pre>
189  *
190  * @return the String representation.
191  */
192
193 public String toString()
194 {
195     return getOwner() + "\t" +
196            getSize() + "\t" +
197            getModDate() + "\t" +
198            getName() + getSuffix();
199 }
200 }

```

```

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

```

```

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

```

```

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

```

```

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

```

```

1 // fo1/10/juno/User.java
2 //
3 //
4 // Copyright 2003 Ethan Bolker and Bill Campbell
5
6 /**
7  * Model a juno user. Each User has a login name, password,
8  * a home directory, and a real name.
9  * name.
10 *
11 * @version 10
12 */
13
14 public class User
15 implements java.io.Serializable
16 {
17     private String name; // The user's login name
18     private String password; // The user's login password.
19     private Directory home; // her home Directory
20     private String realName; // her real name
21
22     /**
23      * Construct a new User.
24      *
25      * @param name the User's login name.
26      * @param password the user's login password.
27      * @param home her home Directory.
28      * @param realName her real name.
29      */
30
31     public User( String name, String password,
32                 Directory home, String realName )
33     {
34         this.name = name;
35         this.password = password;
36         this.home = home;
37         this.realName = realName;
38     }
39
40     /**
41      * Confirm password. Throw a JunoException on failure.
42      *
43      * @param guess the string to test against the password.
44      *
45      * @exception JunoException
46      *         if password fails to match
47      */
48
49     public void matchPassword( String guess ) throws JunoException
50     {
51         if ( !guess.equals( password ) ) {
52             throw new JunoException( "bad password" );
53         }
54     }
55
56     /**

```

```

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

```

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

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

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

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



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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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

```

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

```

```

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

```

```

113 ShellCommandTable table = sh.getSystem().getCommandTable();
114 String [] commandNames = table.getCommandNames();
115 for ( int i = 0; i < commandNames.length; i++ ) {
116     String commandName = commandNames[i];
117     ShellCommand command =
118         table.lookup( commandName );
119
120     CommandMenuAction commandAction =
121         new CommandMenuAction( commandName,
122             new CommandMenuAction( commandName,
123                 command.getArgString() ) );
124
125     HelpMenuAction helpAction =
126         new HelpMenuAction( commandName,
127             command.getArgString(),
128             command.getHelpString() );
129
130     JMenuItem item1 = commandMenu.add( commandAction );
131     JMenuItem item2 = helpMenu.add( helpAction );
132     JButton button = toolbar.add( commandAction );
133     JButton button.setTooltipText( command.getHelpString() );
134
135     }
136
137     this.setMenuBar( menuBar );
138     this.getContentPane().add( toolbar,
139         BorderLayout.NORTH );
140     menuBar.add( commandMenu );
141     menuBar.add( helpMenu );
142     pack();
143     show();
144
145     // add listener to the Do It button
146     doIt.addActionListener( new Interpreter() );
147
148     // add listener to the Logout button and window closer
149     closeMe = new WindowCloser( this );
150     logout.addActionListener( closeMe );
151     this.addWindowListener( closeMe );
152
153     // Set the GUI prompt
154     private void setPrompt( String prompt )
155     {
156         this.promptLabel.setText( prompt );
157     }
158
159     // Implementing the OutputInterface.
160     // Everything goes to the single message area.
161     public void println( String str )
162     {
163         stdout.append( str + "\n" );
164     }

```

```

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

```

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



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

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

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

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