```
 1  // joi/1/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33       * @param amount the amount to be withdrawn.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  // joi/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 the 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       */
56
```



```
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      * The Bank simulation program begins here when the user
133      * issues the command <code>java Bank</code>.
134
135      * @param args the command line arguments (ignored).
136      */
137
138     public static void main( String[] args )
139     {
140         Bank javaBank = new Bank( "Engulf and Devour" );
141         javaBank.open();
142     }
143 }
```

```
 1  // joi/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         Frame frame         = new Frame();
106         TrafficLight light  = new TrafficLight();
107         frame.add( light );
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 WindowAdapter
   121      {
   122           // Close the window by shutting down the light.
   123
   124          public void windowClosing (WindowEvent e)
   125          {
   126              System.exit(0);
   127          }
   128      }
   129  }
   130
   131
```

```
 1  // joi/1/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  // joi/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  // joi/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.fillOval( 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  }
```

```
// joi/1/estore/EStore.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * An EStore object simulates the behavior of a simple on line
 * shopping web site.
 *
 * It contains a Terminal object to model the customer's browser
 * and several Item objects a customer can add to her ShoppingCart.
 *
 * @version 1
 */

public class EStore
{
    private String storeName = "Virtual Minimal Minimall";

    // Use a Terminal object to communicate with customers.
    private Terminal browser = new Terminal();

    // The store stocks two kinds of Items.
    private Item widget = new Item(10); //  widgets cost $10
    private Item gadget = new Item(13); //  gadgets cost $13

    private String selectionList = "(gadget, widget, checkout)";

    /**
     * Visit this EStore.
     *
     * Loop allowing visitor to select items to add to her
     * ShoppingCart.
     */

    public void visit()
    {
        // Create a new, empty ShoppingCart.
        ShoppingCart basket = new ShoppingCart();

        // Print a friendly welcome message.
        browser.println( "Welcome to " + storeName );

        // Change to false when customer is ready to leave:
        boolean stillShopping = true;

        while ( stillShopping ) {
            Item nextPurchase = selectItem();
            if ( nextPurchase == null ) {
                stillShopping = false;
            }
            else {
                basket.add( nextPurchase );
            }
        }
        int numberPurchased = basket.getCount();
```



```
        int totalCost       = basket.getCost();
        browser.println("We are shipping " + numberPurchased + " Items");
        browser.println("and charging your account $" + totalCost);
        browser.println("Thank you for shopping at " + storeName);
    }

    // Discover what the customer wants to do next:
    //  send browser a message to get customer input
    //  examine response to make a choice
    //  if response makes no sense give customer another chance

    private Item selectItem()
    {
        String itemName =
            browser.readWord("Item " + selectionList + ":");

        if ( itemName.equals("widget")) {
            return widget;
        }
        else if ( itemName.equals("gadget")) {
            return gadget;
        }
        else if ( itemName.equals("checkout" )) {
            return null;
        }
        else {
            browser.println( "No item named " +
                             itemName + "; try again" );
            return selectItem();  // try again
        }
    }

    /**
     *  The EStore simulation program begins here when the user
     *  issues the command <code>java EStore</code>.
     */

    public static void main( String[] args )
    {
        // Print this to simulate delay while browser finds store
        System.out.println("connecting ...");

        // Create the EStore object.
        EStore webSite = new EStore();

        // Visit it.
        webSite.visit();
    }
} // end of class EStore
```

```
 1  // joi/1/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  // joi/1/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
61      public int getCost()
62      {
63          return cost;
64      }
65  }
```

```
 1  // joi/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  // joi/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  }
```

```
// joi/2/linear/LinearEquation.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A LinearEquation models equations of the form y = mx + b.
 *
 * @version 2
 */

public class LinearEquation
{
    private double m;      // The equations's slope
    private double b;      // The equations's y-intercept

    /**
     * Construct a LinearEquation from a slope and y-intercept.
     *
     * @param m the slope.
     * @param b the y-intercept.
     */

    public LinearEquation( double m, double b )
    {
        this.m = m;
        this.b = b;
    }

    /**
     * Construct a LinearEquation from two points.
     *
     * @param x1 the x coordinate of the first point
     * @param y1 the y coordinate of the first point
     * @param x2 the x coordinate of the second point
     * @param y2 the y coordinate of the second point
     */

    public LinearEquation( double x1, double y1,
                           double x2, double y2 )
    {
        m = (y2 - y1) / (x2 - x1);
        b = y1 - x1 * m;
    }

    /**
     * Compute y, given x.
     *
     * @param x the input value.
     * @return the corresponding value of y: mx+b.
     */

    public double compute( double x )
    {
        return m*x + b;
    }
```



```

    /**
     * Compute the inverse of this linear equation.
     *
     * @return the LinearEquation object you get by "solving for x".
     */

    public LinearEquation getInverse()
    {
        return new LinearEquation( 1.0/m, -b/m );
    }
}
```

```
 1  // joi/2/arithmetic/IntArithmetic.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 IntArithmetic
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  // joi/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   createDate;  // 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          createDate    = new Date(); // date and time now
42          modDate       = createDate;
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 createDate.toString();
```

```
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.getCreateDate() );
```



```
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  // joi/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  // joi/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+i , 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( "+xxxxxxxxxx          +");
92          terminal.println( "+xxxxx               +");
93          terminal.println( "+                   +");
94          terminal.println( "+   *****           +");
95          terminal.println( "+    1              +");
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  // joi/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
16   *          GGG
17   *          GGG
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( "+RRRR                +");
130         terminal.println( "+RRGGG               +");
131         terminal.println( "+RRGGG               +");
132         terminal.println( "+RRGGG               +");
133         terminal.println( "+  GGRRRRRRR         +");
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  // joi/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  // joi/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          char c = 'a';
21          int x,y;
22          while ( t.readYesOrNo("more")) {
23              char shape = t.readChar("h(line), b(ox), c(lear): ");
24              switch (shape) {
25              case 'h':
26                  int length = t.readInt("HLine length: ");
27                  x          = t.readInt("x coordinate: ");
28                  y          = t.readInt("y coordinate: ");
29                  (new HLine(length, c++)).paintOn(s,x,y);
30                  break;
31              case 'b':
32                  int w = t.readInt("Box width: ");
33                  int h = t.readInt("Box width: ");
34                  x     = t.readInt("x coordinate: ");
35                  y     = t.readInt("y coordinate: ");
36                  (new Box(w,h,c++)).paintOn(s,x,y);
37                  break;
38              case 'c':
39                  s.clear();
40                  break;
41              default:
42                  t.println("try again");
43                  continue;
44              }
45              s.draw(t);
46          }
47      }
48  }
```

```
// joi/3/shapes/TextLine.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

// This file contains stubs for the methods.

/**
 * A horizontal line of character text.
 *
 * @version 3
 */

public class TextLine
{
    /**
     * Construct a TextLine.
     *
     * @param text the text of the line.
     */

    public TextLine( String text )
    {
    }

    /**
     * Paint this TextLine on Screen s at position (x,y).
     *
     * @param s the Screen on which this line is to be painted.
     * @param x the x position for the line.
     * @param y the y position for the line.
     */

    public void paintOn( Screen s, int x, int y )
    {
    }

    /**
     * Draw the TextLine to Screen s at position (0,0).
     *
     * @param s the Screen on which this line is to be painted.
     */

    public void paintOn( Screen s )
    {
        paintOn( s, 0, 0 );
    }

    /**
     * Get the length of this line.
     *
     * @return the length in (character) pixels.
     */

    public int getLength()
    {
```



```
        return 0; // replace with the right answer
    }

    /**
     * Unit test for class.TextLine,
     * assuming Screen and Terminal work.
     */

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

```
 1  // joi/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).
56       *
```



```
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);                        // left edge
107             for ( int col = 0; col < width; col++ ) {
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 }
```

```
// joi/4/bank/Bank.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

// Lines marked "///" flag places where changes will be needed.

/// import java.util.??

/**
 * A Bank object simulates the behavior of a simple bank/ATM.
 * It contains a Terminal object and a collection of
 * BankAccount objects.
 *
 * Its public method visit opens the this Bank for business,
 * prompting the customer for input.
 *
 * To create a Bank and open it for business issue the command
 * <code>java Bank</code>.
 *
 * @see BankAccount
 * @version 4
 */

public class Bank
{
    private String bankName;           // the name of this Bank
    private Terminal atm;              // for talking with the customer
    private int balance = 0;           // total cash on hand
    private int transactionCount = 0;  // number of Bank transactions don

    private BankAccount[] accountList; /// collection of BankAccounts
    /// omit next line when accountList is dynamic
    private final static int NUM_ACCOUNTS = 3;

    // what the banker can ask of the bank

    private static final String BANKER_COMMANDS =
    "Banker commands: " +
        "exit, open, customer, report, help.";

    // what the customer can ask of the bank

    private static final String CUSTOMER_TRANSACTIONS =
    "    Customer transactions: " +
        "deposit, withdraw, transfer, balance, quit, help.";

    /**
     * Construct a Bank with the given name and Terminal.
     *
     * @param bankName the name for this Bank.
     * @param atm  this Bank's Terminal.
     */

    public Bank( String bankName, Terminal atm )
    {
```



```
        this.atm      = atm;
        this.bankName = bankName;
        // initialize collection:
        accountList   = new BankAccount[NUM_ACCOUNTS]; ///

        /// When accountList is an array, fill it here.
        /// When it's an ArrayList or a TreeMap, delete these lines.
        /// Bank starts with no accounts, banker creates them with
        /// the openNewAccount method.
        accountList[0] = new BankAccount(  0, this);
        accountList[1] = new BankAccount(100, this);
        accountList[2] = new BankAccount(200, this);
    }

    /**
     * Simulates interaction with a Bank.
     * Presents the user with an interactive loop, prompting for
     * banker transactions and in case of the banker transaction
     * "customer", an account id and further customer
     * transactions.
     */

    public void visit()
    {
        instructUser();

        String command;
        while (!(command =
                 atm.readWord("banker command: ")).equals("exit")) {

            if (command.startsWith("h")) {
                help( BANKER_COMMANDS );
            }
            else if (command.startsWith("o")) {
                openNewAccount();
            }
            else if (command.startsWith("r")) {
                report();
            }
            else if (command.startsWith( "c" ) ) {
                BankAccount acct = whichAccount();
                if ( acct != null )
                    processTransactionsForAccount( acct );
            }
            else {
                // Unrecognized Request
                atm.println( "unknown command: " + command );
            }
        }
        report();
        atm.println( "Goodbye from " + bankName );
    }


    // Open a new bank account,
    // prompting the user for information.
```

```
113
114     private void openNewAccount()
115     {
116         /// when accountList is a dynamic collection
117         /// remove the next two lines, uncomment and complete
118         /// the code between /* and */
119         atm.println(bankName + " is accepting no new customers\n");
120         return;
121
122         /*
123         // prompt for initial deposit
124         int startup = atm.readInt( "Initial deposit: " );
125
126         // create newAccount
127         BankAccount newAccount = new BankAccount( startup, this );
128
129         //  and add it to accountList
130         ???
131
132         // inform user
133         atm.println( "opened new account " + ??? /// name or number
134                         + " with $" + newAccount.getBalance());
135         */
136     }
137
138     // Prompt the customer for transaction to process.
139     // Then send an appropriate message to the account.
140
141     private void processTransactionsForAccount( BankAccount acct )
142     {
143         help( CUSTOMER_TRANSACTIONS );
144
145         String transaction;
146         while (!(transaction =
147                  atm.readWord("    transaction: ")).equals("quit")) {
148
149             if ( transaction.startsWith( "h" ) ) {
150                 help( CUSTOMER_TRANSACTIONS );
151             }
152             else if ( transaction.startsWith( "d" ) ) {
153                 int amount = atm.readInt( "    amount:" );
154                 atm.println("    deposited " + acct.deposit( amount ));
155             }
156             else if ( transaction.startsWith( "w" ) ) {
157                 int amount = atm.readInt( "    amount:" );
158                 atm.println("    withdrew " + acct.withdraw( amount ));
159             }
160             else if (transaction.startsWith("t")) {
161                 atm.print( "    to ");
162                 BankAccount toacct = whichAccount();
163                 if (toacct != null) {
164                     int amount = atm.readInt("    amount to transfer: ");
165                     atm.println("    transfered " +
166                                 toacct.deposit(acct.withdraw(amount)));
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
180     // Prompt for an account name (or number), look it up
181     // in the account list. If it's there, return it;
182     // otherwise report an error and return null.
183
184     private BankAccount whichAccount()
185     {
186         /// prompt for account name or account number
187         /// (whichever is appropriate)
188         int accountNumber = atm.readInt("account number: ");
189
190         /// look up account in accountList
191         /// if it's there, return it
192         /// else the following two lines should execute
193         if ( accountNumber >= 0 && accountNumber < NUM_ACCOUNTS ) {
194             return accountList[accountNumber];
195         }
196         else {
197             atm.println("not a valid account");
198             return null;
199         }
200     }
201
202     // Report bank activity.
203     // For each BankAccount, print the customer id (name or number),
204     // account balance and the number of transactions.
205     // Then print Bank totals.
206
207     private void report()
208     {
209         atm.println( "\nSummaries of individual accounts:" );
210         atm.println( "account  balance   transaction count" );
211         for (int i = 0; i < NUM_ACCOUNTS; i++ ) {             ///
212             atm.println(i + "\t" + accountList[i].getBalance() +     ///
213                         "\t" + accountList[i].getTransactionCount());///
214         }
215
216         atm.println( "\nBank totals");
217         atm.println( "open accounts: " + getNumberOfAccounts() );
218         atm.println( "cash on hand: $" + getBalance());
219         atm.println( "transactions:  " + getTransactionCount());
220         atm.println();
221     }
222
223
224     // Welcome the user to the bank and instruct her on
```

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



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

```
// joi/4/bank/BankAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A BankAccount object has private fields to keep track
 * of its current balance, the number of transactions
 * performed and the Bank in which it is an account, and
 * and public methods to access those fields appropriately.
 *
 * @see Bank
 * @version 4
 */

public class BankAccount
{
    private int  balance = 0;          // Account balance (whole dollars)
    private int  transactionCount = 0; // Number of transactions performe
    private Bank issuingBank;          // Bank issuing this account

    /**
     * Construct a BankAccount with the given initial balance and
     * issuing Bank. Construction counts as this BankAccount's
     * first transaction.
     *
     * @param initialBalance the opening balance.
     * @param issuingBank the bank that issued this account.
     */

    public BankAccount( int initialBalance, Bank issuingBank )
    {
        this.issuingBank = issuingBank;
        deposit( initialBalance );
    }

    /**
     * Withdraw the given amount, decreasing this BankAccount's
     * balance and the issuing Bank's balance.
     * Counts as a transaction.
     *
     * @param amount the amount to be withdrawn
     * @return amount withdrawn
     */

    public int withdraw( int amount )
    {
        incrementBalance( -amount );
        countTransaction();
        return amount ;
    }

    /**
     * Deposit the given amount, increasing this BankAccount's
     * balance and the issuing Bank's balance.
     * Counts as a transaction.
```



```
     *
     * @param amount the amount to be deposited
     * @return amount deposited
     */

    public int deposit(int amount)
    {
        incrementBalance( amount);
        countTransaction();
        return amount ;
    }

    /**
     * Request for balance. Counts as a transaction.
     *
     * @return current account balance
     */

    public int requestBalance()
    {
        countTransaction();
        return getBalance() ;
    }

    /**
     * Get the current balance.
     * Does NOT count as a transaction.
     *
     * @return current account balance
     */

    public int getBalance()
    {
        return balance;
    }

    /**
     * Increment account balance by given amount.
     * Also increment issuing Bank's balance.
     * Does NOT count as a transaction.
     *
     * @param amount the amount increment.
     */

    public void incrementBalance( int amount )
    {
        balance += amount;
        this.getIssuingBank().incrementBalance( amount );
    }

    /**
     * Get the number of transactions performed by this
     * account. Does NOT count as a transaction.
     *
     * @return number of transactions performed.
     */
```

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

```
Welcome to River Bank
Open some accounts and work with them.
Banker commands: exit, open, customer, report, help.

banker command: open
Initial deposit: 1000
opened new account 0 with $1000
banker command: open
Initial deposit: 2000
opened new account 1 with $2000
banker command: help
Banker commands: exit, open, customer, report, help.

banker command: report

Summaries of individual accounts:
account  balance   transaction count
0       $1000           1
1       $2000           1

Bank totals
open accounts: 2
cash on hand: $3000
transactions:  2

banker command: open
Initial deposit: 3000
opened new account 2 with $3000
banker command: customer
account number: 0
    Customer transactions: deposit, withdraw, transfer, balance, quit, he

    transaction: balance
    current balance 1000
    transaction: deposit
    amount:9999
    deposited 9999
    transaction: balance
    current balance 10999
    transaction: quit

banker command: customer
account number: 1
    Customer transactions: deposit, withdraw, transfer, balance, quit, he

    transaction: transfer
    to account number: 9
not a valid account
    transaction: transfer
    to account number: 2
    amount to transfer: 45
    transfered 45
    transaction: quit

banker command: exit

```



```
Summaries of individual accounts:
account  balance   transaction count
0       $10999          4
1       $1955           2
2       $3045           2

Bank totals
open accounts: 3
cash on hand: $15999
transactions:  8

Goodbye from River Bank
```

```
 1  open
 2  groucho
 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  groucho
16  balance
17  deposit
18  9999
19  balance
20  quit
21  customer
22  harpo
23  transfer
24  chico
25  45
26  quit
27  exit
```

```
Welcome to River Bank
Open some accounts and work with them.
Banker commands: exit, open, customer, report, help.

banker command: open
Account name: groucho
Initial deposit: 1000
opened new account groucho with $1000
banker command: customer
account name: harpo
not a valid account
banker command: open
Account name: harpo
Initial deposit: 2000
opened new account harpo with $2000
banker command: help
Banker commands: exit, open, customer, report, help.

banker command: report

Summaries of individual accounts:
account  balance   transaction count
groucho $1000           1
harpo   $2000           1

Bank totals
open accounts: 2
cash on hand: $3000
transactions:  2

banker command: open
Account name: chico
Initial deposit: 3000
opened new account chico with $3000
banker command: customer
account name: groucho
    Customer transactions: deposit, withdraw, transfer, balance, quit, he

    transaction: balance
    current balance 1000
    transaction: deposit
    amount:9999
    deposited 9999
    transaction: balance
    current balance 10999
    transaction: quit

banker command: customer
account name: harpo
    Customer transactions: deposit, withdraw, transfer, balance, quit, he

    transaction: transfer
    to account name: chico
    amount to transfer: 45
    transfered 45
    transaction: quit
```



```

banker command: exit

Summaries of individual accounts:
account  balance   transaction count
chico   $3045           2
groucho $10999          4
harpo   $1955           2

Bank totals
open accounts: 3
cash on hand: $15999
transactions:  8

Goodbye from River Bank
```

```
 1  // joi/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  }
```

```
// joi/4/dictionary/Dictionary.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * Model a dictionary with a TreeMap of (word, Definition) pairs.
 *
 * @see Definition
 *
 * @version 4
 */

public class Dictionary
{
    private TreeMap entries;

    /**
     * Construct an empty Dictionary.
     */

    public Dictionary()
    {
        entries = new TreeMap();
    }

    /**
     * Add an entry to this Dictionary.
     *
     * @param word the word being defined.
     * @param definition the Definition of that word.
     */

    public void addEntry( String word, Definition definition )
    {
        entries.put( word, definition );
    }

    /**
     * Look up an entry in this Dictionary.
     *
     * @param word the word whose definition is sought
     * @return the Definition of that word, null if none.
     */

    public Definition getEntry( String word )
    {
        return (Definition)entries.get(word);
    }

    /**
     * Get the size of this Dictionary.
     *
     * @return the number of words.
```



```
     */

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

    /**
     * Construct a String representation of this Dictionary.
     *
     * @return a multiline String representation.
     */

    public String toString()
    {
        String str = "";
        String word;
        Definition definition;
        Set allWords = entries.keySet();
        Iterator wordIterator = allWords.iterator();
        while ( wordIterator.hasNext() ) {
            word = (String)wordIterator.next();
            definition = this.getEntry( word );
            str += word + ":\n" + definition.toString() + "\n";
        }
        return str;
    }
}
```

```
 1  // joi/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  }
```

```
// joi/4/dictionary/Lookup.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * On line word lookup.
 *
* @see Dictionary
 * @see Definition
 *
 * @version 4
 */

public class Lookup
{
    private static Terminal t = new Terminal();
    private static Dictionary dictionary = new Dictionary();

    /**
     * Helper method to fill the dictionary with some simple
     * Definitions.
     *
     * A real Dictionary would live in a file somewhere.
     */

    private static void fillDictionary()
    {
        dictionary.addEntry( "shape",
            new Definition(  "a geometric object in a plane" ) );
        dictionary.addEntry( "quadrilateral",
            new Definition(  "a polygonal shape with four sides" ) );
        dictionary.addEntry( "rectangle",
            new Definition(  "a right-angled quadrilateral" ) );
        dictionary.addEntry( "square",
            new Definition(  "a rectangle having equal sides" ) );
    }

    /**
     * Helper method to print the Definition of a single word,
     * or a message if the word is not in the Dictionary.
     *
     * @param word the word whose definition is wanted.
     */

    private static void printDefinition(String word)
    {
        Definition definition = dictionary.getEntry(word);
        if (definition == null) {
            t.println("sorry, no definition found for " + word);
        }
        else {
            t.println(definition.toString());
        }
    }

```



```
    /**
     * Run the Dictionary lookup.
     *
     * Parse command line arguments for words to look up,
     * "all" prints the whole Dictionary.
     *
     * Then prompt for more words, "quit" to finish.
     *
     * For example,
     * <pre>
     *
     * %> java Lookup shape square circle
     * shape:
     * a geometric object in a plane
     * square:
     * a rectangle having equal sides
     * circle:
     * sorry, no definition found for circle
     *
     * look up words, "quit" to quit
     * word> rectangle
     * a right-angled quadrilateral
     * word> quit
     * %>
     * </pre>
     *
     * @param args the words that we want looked up, supplied as
     *             command line arguments.  If the word "all" is
     *             included, all words are looked up.
     */

    public static void main( String[] args )
    {
        // fill the dictionary (not a big one!)
        fillDictionary();

        // look up some words
        String word;

        // words specified on command line
        for (int i = 0; i < args.length; i++ ) {
            word = args[i];
            if (word.equals("all")) {
                t.println("The whole dictionary (" +
                          dictionary.getSize() + " entries):");
                t.println("-------------------");
                t.println(dictionary.toString());
                t.println("-------------------");
            }
            else {
                t.println(word + ":");
                printDefinition(word);
            }
        }

        // words entered interactively
```

```
   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  // joi/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   createDate;  // 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          createDate    = new Date(); // date and time now
42          modDate       = createDate;
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 createDate.toString();
```

```
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.getCreateDate() );
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  }
```

```
// joi/4/textfiles/Directory.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

// This draft contains just stubs for the methods.
// You can invoke them all, but none will do anything.

/**
 * Directory of TextFiles.
 *
 * @version 4
 */

public class Directory
{
    /**
     * Construct a Directory.
     */

    public Directory( )
    {
    }

    /**
     * The size of a directory is the number of TextFiles it contains.
     *
     * @return the number of TextFiles.
     */

    public int getSize()
    {
        return 0;
    }

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

    public void addTextFile(String name, TextFile afile)
    {
    }

    /**
     * Get a TextFile in this Directory, by name .
     *
     * @param filename the name of the TextFile to find.
     * @return the TextFile found, null if none.
     */

    public TextFile retrieveTextFile( String filename )
    {
```



```
        return null;
    }

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

    public String[] getFileNames()
    {
        // pseudocode for an implementation:
        // declare an array of String
        // create that array with as many spaces as there
        //   are TextFile's in this Directory
        // loop through the keys of the TreeMap of TextFiles,
        // adding each String key to the array
        // return the array

        // the next line is there because we have to return
        // _something_ in order to satisfy the compiler
        return new String[0];
    }

    /**
     *  main, for unit testing.
     *
     * The command
     * <pre>
     *   java Directory
     * </pre>
     * should produce output
     * <pre>
     * bill     17      Sun Jan 06 19:40:13 EST 2003    diary
     * eb       12      Sun Jan 06 19:40:13 EST 2003    greeting
     * </pre>
     * (with current dates, of course).
     */

    public static void main( String[] args )
    {
        Directory dir = new Directory();
        dir.addTextFile("greeting", new TextFile("eb", "Hello, world"));
        dir.addTextFile("diary", new TextFile("bill", "Writing Directory"
        // now list TextFiles in dir to get output specified
    }
}
```

```
// joi/4/estore/EStore.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * An EStore object simulates the behavior of a simple on line
 * shopping web site.
 *
 * It contains a Terminal object to model the customer's browser
 * and a Catalog of Items that may be purchased and
 * then added to the customer's shoppingCart.
 *
 * @version 4
 */

public class EStore
{
    private String   storeName;
    private Terminal browser;
    private Catalog  catalog;

    /**
     *  Construct a new EStore.
     *
     *  @param storeName the name of the EStore
     *  @param browser the visitor's Terminal.
     */

    public EStore( String storeName, Terminal browser )
    {
        this.browser   = browser;
        this.storeName = storeName;
        this.catalog   = new Catalog();
        catalog.addItem( new Item("quaffle", 55) );
        catalog.addItem( new Item("bludger", 15) );
        catalog.addItem( new Item("snitch",  1000) );
    }

    /**
     *  Visit this EStore.
     *
     *  Execution starts here when the store opens for
     *  business. User can visit as a customer, act as
     *  the manager, or exit.
     */

    public void visit()
    {
        // Print a friendly welcome message.
        browser.println( "Welcome to " + storeName );
        while (true) { // an infinite loop ...
            browser.println();
            String whoAreYou = browser.readWord(
               storeName + " (manager, visit, exit): ");
            if (whoAreYou.equals("exit")) {
```



```
                break; // leave the while loop
            }
            if (whoAreYou.equals("manager")) {
                managerVisit();
            }
            if (whoAreYou.equals("visit")) {
                customerVisit();
            }
        }
    }

    /**
     *  Manager options:
     *
     *  examine the catalog
     *  add an Item to the catalog
     *  quit
     */
    private void managerVisit( )
    {
        while (true) {
            String cmd =
              browser.readWord("manager command (show, new, quit):");
            if (cmd.equals("quit")) {
                break;  // leave manager command while loop
            }
            else if (cmd.equals("show")) {
                catalog.show(browser);
            }
            else if (cmd.equals("new")) {
                String itemName = browser.readWord("  item name:     ");
                int cost        = browser.readInt( "  cost:          ");
                catalog.addItem( new Item(itemName, cost) );
            }
            else {
                browser.println("unknown manager command: " + cmd);
            }
        }
    }

    /**
     * Customer visits this EStore.
     *
     * Loop allowing customer to select items to add to her
     * shoppingCart.
     */

    private void customerVisit( )
    {
        // Create a new, empty ShoppingCart.
        ShoppingCart basket = new ShoppingCart();

        browser.println( "Currently available:");
        catalog.show(browser);
        while ( true ) { // loop forever ...
            String nextPurchase = browser.readWord(
```

```
113                     "select your purchase, checkout, help: ");
114
115             if ( nextPurchase.equals("checkout" )) break; // leave loop!
116
117             if ( nextPurchase.equals("help" )) {
118                 catalog.show(browser);
119                 continue; // go back to top of while loop
120             }
121             // customer has entered the name of an Item
122             basket.addItem( catalog.getItem(nextPurchase) );
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 Minimall"; //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  // joi/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       * Items it contains.
58       *
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  // joi/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  // joi/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  }
```

```
// joi/5/shapes/Line.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A Line has a length and a paintChar used to paint
 * itself on a Screen.
 *
 * Subclasses of this abstract class specify the direction
 * of the Line.
 *
 * @version 5
 */

public abstract class Line
{
    protected int  length;      // length in (character) pixels.
    protected char paintChar;   // character used for painting.

    /**
     * Construct a Line.
     *
     * @param length length in (character) pixels.
     * @param paintChar character used for painting this Line.
     */

    protected Line( int length, char paintChar )
    {
        this.length    = length;
        this.paintChar = paintChar;
    }

    /**
     * Get the length of this line.
     *
     * @return the length in (character) pixels.
     */

    public int getLength()
    {
        return length;
    }

    /**
     * Set the length of this line.
     *
     * @param length the new length in (character) pixels.
     */

    public void setLength( int length )
    {
        this.length = length;
    }

    /**
```



```
     * Get the paintChar of this Line.
     *
     * @return the paintChar.
     */

    public char getPaintChar()
    {
        return paintChar;
    }

    /**
     * Set the paintChar of this Line.
     *
     * @param paintChar the new paintChar.
     */

    public void setPaintChar( char paintChar )
    {
        this.paintChar = paintChar;
    }

    /**
     * Paint this Line on Screen s at position (x,y).
     *
     * @param s the Screen on which this Line is to be painted.
     * @param x the x position for the line.
     * @param y the y position for the line.
     */

    public abstract void paintOn( Screen s, int x, int y );

    /**
     * Paint this Line on Screen s at position (0,0).
     *
     * @param s the Screen on which this Line is to be painted.
     */

    public void paintOn( Screen s )
    {
        paintOn( s, 0, 0 );
    }
}
```

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



```
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  // joi/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  }
```

```
// joi/5/shapes/ShapeOnScreen.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

// This file is used in one of the Chapter 5 exercises on shapes.

/**
 * A ShapeOnScreen models a Shape to be painted at
 * a given position on a Screen.
 *
 * @see Shape
 * @see Screen
 *
 * @version 5
 */

public class ShapeOnScreen
{
    private Shape shape;
    private int x;
    private int y;

    /**
     * Construct a ShapeOncreen.
     *
     * @param shape the Shape
     * @param x its x coordinate
     * @param y its y coordinate
     */

    public ShapeOnScreen( Shape shape, int x, int y )
    {
        this.shape = shape;
        this.x     = x;
        this.y     = y;
    }

    /**
     * What Shape does this ShapeOnScreen represent?
     *
     * @return the Shape.
     */

    public Shape getShape() {
        return shape;
    }

    /**
     * The current x coordinate of this ShapeOnScreen.
     *
     * @return the x coordinate.
     */

    public int getX() {
        return x;
```



```
    }

    /**
     * The current y coordinate of this ShapeOnScreen.
     *
     * @return the y coordinate.
     */

    public int getY() {
        return y;
    }

    /**
     * Unit test.
     */

    public static void main( String[] args ) {
        ShapeOnScreen sos = new ShapeOnScreen( null, 5, 7);
        System.out.println("Shape: " + sos.getShape());
        System.out.println("x:     " + sos.getX());
        System.out.println("y:     " + sos.getY());
    }
}
```

```
// joi/5/jfiles/JFile.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

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

/**
 * A JFile object models a file in a hierarchical file system.
 * <p>
 * Extend this abstract class to create particular kinds of JFiles,
 * e.g.:<br>
 * Directory  -
 *    a JFile that maintains a list of the files it contains.<br>
 * TextFile  -
 *    a JFile containing text you might want to read.<br>
 *
 * @see Directory
 * @see TextFile
 *
 * @version 5
 */

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

    public static final String separator = File.separator;

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

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

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



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

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

    public String getName()
    {
        return name;
    }

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

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

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

    public abstract int getSize();

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

    public abstract String getSuffix();

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

    public void setOwner( String 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 String 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
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      // display the contents of a TextFile
272
273      private static void type( TextFile file )
274      {
275         String whatToPrint;
276         if (file == null) {
277             whatToPrint = "no such file";
278         }
279         else {
280             whatToPrint = file.getContents();
```



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

```
// joi/5/jfiles/Directory.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Directory of JFiles.
 *
 * A Directory is a JFile that maintains a
 * table of the JFiles it contains
 *
 * @version 5
 */

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

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

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

    /**
     * The size of a directory is the number of TextFiles it contains.
     *
     * @return the number of TextFiles.
     */

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

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

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



```
    }

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

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

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

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

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

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

```
// joi/5/jfiles/TextFile.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * A TextFile is a JFile that holds text.
 *
 * @version 5
 */

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

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

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

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

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

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

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

    /**
```



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

    public String getSuffix()
    {
        return "";
    }

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

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

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

    public String getContents()
    {
        return contents;
    }

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

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


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

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

```
// joi/5/bank/Bank.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * A Bank object simulates the behavior of a simple bank/ATM.
 * It contains a Terminal object and a collection of
 * BankAccount objects.
 *
 * The visit method opens this Bank for business,
 * prompting the customer for input.
 *
 * To create a Bank and open it for business issue the command
 * <code>java Bank</code>.
 *
 * @see BankAccount
 * @version 5
 */

public class Bank
{
    private String bankName;           // the name of this Bank
    private Terminal atm;              // for talking with the customer
    private int balance = 0;           // total cash on hand
    private int transactionCount = 0;  // number of Bank transactions
    private Month month;               // the current month.

    private TreeMap accountList;      // mapping names to accounts.

    // what the banker can ask of the bank

    private static final String BANKER_COMMANDS =
    "Banker commands: " +
        "exit, open, customer, report, help.";

    // what the customer can ask of the bank

    private static final String CUSTOMER_TRANSACTIONS =
    "    Customer transactions: " +
    "deposit, withdraw, transfer, balance, cash check, quit, help.";

    /**
     * Construct a Bank with the given name and Terminal.
     *
     * @param bankName the name for this Bank.
     * @param atm  this Bank's Terminal.
     */

    public Bank( String bankName, Terminal atm )
    {
        this.atm      = atm;
        this.bankName = bankName;
        accountList   = new TreeMap();
```



```
        month         = new Month();
    }

    /**
     * Simulates interaction with a Bank.
     * Presents the user with an interactive loop, prompting for
     * banker transactions and in case of the banker transaction
     * "customer", an account id and further customer
     * transactions.
     */

    public void visit()
    {
        instructUser();

        String command;
        while (!(command =
                 atm.readWord("banker command: ")).equals("exit")) {

            if (command.startsWith("h")) {
                help( BANKER_COMMANDS );
            }
            else if (command.startsWith("o")) {
                openNewAccount();
            }
            else if (command.startsWith("r")) {
                report();
            }
            else if (command.startsWith( "c" ) ) {
                BankAccount acct = whichAccount();
                if ( acct != null )
                    processTransactionsForAccount( acct );
            }
            else {
                // Unrecognized Request
                atm.println( "unknown command: " + command );
            }
        }
        report();
        atm.println( "Goodbye from " + bankName );
    }


    // Open a new bank account,
    // prompting the user for information.

    private void openNewAccount()
    {
        String accountName = atm.readWord( "Account name: " );
        char accountType =
            atm.readChar( "Checking/Fee/Regular? (c/f/r): " );
        int startup = atm.readInt( "Initial deposit: " );
        BankAccount newAccount;
        switch( accountType ) {
        case 'c':
            newAccount = new CheckingAccount( startup, this );
```

```
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
136         String transaction;
137         while (!(transaction =
138                  atm.readWord("    transaction: ")).equals("quit")) {
139
140             if ( transaction.startsWith( "h" ) ) {
141                 help( CUSTOMER_TRANSACTIONS );
142             }
143             else if ( transaction.startsWith( "d" ) ) {
144                 int amount = atm.readInt( "    amount:" );
145                 atm.println("    deposited " + acct.deposit( amount ));
146             }
147             else if ( transaction.startsWith( "w" ) ) {
148                 int amount = atm.readInt( "    amount:" );
149                 atm.println("    withdrew " + acct.withdraw( amount ));
150             }
151             else if ( transaction.startsWith( "c" ) ) {
152                 int amount = atm.readInt( "    amount of check: " );
153                 atm.println("    cashed check for " +
154                             ((CheckingAccount)acct).honorCheck( amount ))
155             }
156             else if (transaction.startsWith("t")) {
157                 atm.print( "    to ");
158                 BankAccount toacct = whichAccount();
159                 if (toacct != null) {
160                     int amount = atm.readInt("    amount to transfer: ");
161                     atm.println("    transfered " +
162                                 toacct.deposit(acct.withdraw(amount)));
163                 }
164             }
165             else if (transaction.startsWith("b")) {
166                 atm.println("    current balance " +
167                             acct.requestBalance());
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       *
247       * @param amount the amount increment.
248       */
249
250      public void incrementBalance(int amount)
251      {
252          balance += amount;
253      }
254
255      /**
256       * Increment by one the count of transactions,
257       * for this bank.
258       */
259
260      public void countTransaction()
261      {
262          transactionCount++;
263      }
264
265      /**
266       * Get the number of transactions performed by this bank.
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       *
279       * @return current bank balance.
280       */
```



```
281
282      public int getBalance()
283      {
284          return balance;
285      }
286
287      /**
288       * Get the current number of open accounts.
289       *
290       * @return number of open accounts.
291       */
292
293      public int getNumberOfAccounts()
294      {
295          return accountList.size();
296      }
297
298      /**
299       * Run the simulation by creating and then visiting a new Bank.
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 = "Faithless 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  }
```

```
// joi/5/bank/BankAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A BankAccount object has private fields to keep track
 * of its current balance, the number of transactions
 * performed and the Bank in which it is an account, and
 * and public methods to access those fields appropriately.
 *
 * @see Bank
 * @version 5
 */

public abstract class BankAccount
{
    private int  balance = 0;          // Account balance (whole dollars)
    private int  transactionCount = 0; // Number of transactions performe
    private Bank issuingBank;          // Bank issuing this account

    /**
     * Construct a BankAccount with the given initial balance and
     * issuing Bank. Construction counts as this BankAccount's
     * first transaction.
     *
     * @param initialBalance the opening balance.
     * @param issuingBank the bank that issued this account.
     */

    public BankAccount( int initialBalance, Bank issuingBank )
    {
        this.issuingBank = issuingBank;
        deposit( initialBalance );
    }

    /**
     * Withdraw the given amount, decreasing this BankAccount's
     * balance and the issuing Bank's balance.
     * Counts as a transaction.
     *
     * @param amount the amount to be withdrawn
     * @return amount withdrawn
     */

    public int withdraw( int amount )
    {
        incrementBalance( -amount );
        countTransaction();
        return amount ;
    }

    /**
     * Deposit the given amount, increasing this BankAccount's
     * balance and the issuing Bank's balance.
     * Counts as a transaction.
```



```
     *
     * @param amount the amount to be deposited
     * @return amount deposited
     */

    public int deposit(int amount)
    {
        incrementBalance( amount);
        countTransaction();
        return amount ;
    }

    /**
     * Request for balance. Counts as a transaction.
     *
     * @return current account balance.
     */

    public int requestBalance()
    {
        countTransaction();
        return getBalance() ;
    }

    /**
     * Get the current balance.
     * Does NOT count as a transaction.
     *
     * @return current account balance
     */

    public int getBalance()
    {
        return balance;
    }

    /**
     * Increment account balance by given amount.
     * Also increment issuing Bank's balance.
     * Does NOT count as a transaction.
     *
     * @param amount the amount of the increment.
     */

    public void incrementBalance( int amount )
    {
        balance += amount;
        this.getIssuingBank().incrementBalance( amount );
    }

    /**
     * Get the number of transactions performed by this
     * account. Does NOT count as a transaction.
     *
     * @return number of transactions performed.
     */
```

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

```
 1  // joi/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  // joi/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  // joi/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  // joi/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  //      public String toString()
54  //      {
55  //      }
56
```



```
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  }
```

```
//  joi/6/juno/Juno.java
//
//
//  Copyright 2003 Bill Campbell and Ethan Bolker

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

/**
 * Juno (Juno's Unix NOt) mimics a command line operating system
 * like Unix.
 * <p>
 * A Juno system has a name, a set of Users, a JFile system,
 * a login process and a set of shell commands.
 *
 * @see User
 * @see JFile
 * @see ShellCommand
 *
 * @version 6
 */

public class Juno
{
    private final static String os      = "Juno";
    private final static String version = "6";

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

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

    private ShellCommandTable commandTable; // shell commands

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

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

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

        // the file system

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



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

        // create, then start a command line login interpreter

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

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

    public String getHostName()
    {
        return hostName;
    }

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

    public String getOS()
    {
        return os;
    }

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

    public String getVersion()
    {
        return version;
    }

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

    public Directory getUserHomes()
    {
        return userHomes;
    }
```

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

```
// joi/6/juno/LoginInterpreter.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Interpreter for Juno login commands.
 *
 * There are so few commands that if-then-else logic is OK.
 *
 * @version 6
 */

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

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

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

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

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

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

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

    public void CLILogin()
    {
```



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

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

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

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

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

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

```

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

```
// joi/6/juno/Shell.java
//
//
// Copyright 2003, Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Models a shell (command interpreter)
 *
 * The Shell knows the (Juno) system it's working in,
 * the User who started it,
 * and the console to which to send output.
 *
 * It keeps track of the the current working directory (.) .
 *
 * @version 6
 */

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

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

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

    // Run the command line interpreter

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

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



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

    private boolean interpret( String inputLine )
    {
        StringTokenizer st = stripComments(inputLine);
        if (st.countTokens() == 0) {
            return true;            // skip blank line
        }
        String commandName = st.nextToken();
        if (commandName.equals( "logout" )) {
            return false;           // user is done
        }
        ShellCommand commandObject =
            system.getCommandTable().lookup( commandName );
        if (commandObject == null ) {
            console.errPrintln( "Unknown command: " + commandName );
        }
        else {
            commandObject.doIt( st, this );
        }
        return true;
    }

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

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

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

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

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

    public User getUser()
    {
        return user;
```

```
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  }
```

```
// joi/6/juno/ShellCommand.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Model those features common to all ShellCommands.
 *
 * Each concrete extension of this class provides a constructor
 * and an implementation for method doIt.
 *
 * @version 6
 */

public abstract class ShellCommand
{
    private String helpString;   // documents the command
    private String argString;    // any args to the command

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

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

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

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

    /**
     * Execute the command.
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     */

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

    /**
```



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

    public String getHelpString()
    {
        return helpString;
    }

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

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

```
 1  // joi/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  // joi/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 reminder 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                retrieveJFile( filename ) ).getContents() );
43      }
44  }
```

```
 1  // joi/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 doIt( 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  // joi/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  }
```

```
// joi/6/juno/ShellCommandTable.java (version 6)
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * A ShellCommandTable object maintains a dispatch table of
 * ShellCommand objects keyed by the command names used to invoke
 * them.
 *
 * To add a new shell command to the table, install it from
 * method fillTable().
 *
 * @see ShellCommand
 *
 * @version 6
 */

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

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

    public ShellCommandTable()
    {
        fillTable();
    }

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

    public ShellCommand lookup( String key )
    {
        return (ShellCommand)table.get( key );
    }

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

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



```
    }

    // Associate a command name with a ShellCommand.

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

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

    private void fillTable()
    {
        install( "newfile", new NewfileCommand() );
        install( "type", new TypeCommand() );
        install( "mkdir", new MkdirCommand() );
        install( "help", new HelpCommand() );
    }
}
```

```
 1  // joi/6/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 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  }
```

```
// joi/6/jfiles/Directory.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Directory of JFiles.
 *
 * A Directory is a JFile that maintains a
 * table of the JFiles it contains
 *
 * @version 6
 */

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

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

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

    /**
     * The size of a directory is the number of TextFiles it contains.
     *
     * @return the number of TextFiles.
     */

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

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

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



```
    }

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

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

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

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

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

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

```
// joi/6/jfiles/TextFile.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * A TextFile is a JFile that holds text.
 *
 * @version 6
 */

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

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

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

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

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

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

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

    /**
```



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

    public String getSuffix()
    {
        return "";
    }

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

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

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

    public String getContents()
    {
        return contents;
    }

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

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


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

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

```
 1  // joi/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  // joi/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 maxFreeTransactions = 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
73      public void visit()
74      {
75          instructUser();
76
77          String command;
78          while (!(command =
79                   atm.readWord("banker command: ")).equals("exit")) {
80
81              if (command.startsWith("h")) {
82                  help( BANKER_COMMANDS );
83              }
84              else if (command.startsWith("o")) {
85                  openNewAccount();
86              }
87              else if (command.startsWith("n")) {
88                  newMonth();
89              }
90              else if (command.startsWith("r")) {
91                  report();
92              }
93              else if (command.startsWith( "c" ) ) {
94                  BankAccount acct = whichAccount();
95                  if ( acct != null ) {
96                          processTransactionsForAccount( acct );
97                  }
98              }
99              else {
100                 // Unrecognized Request
101                 atm.println( "unknown command: " + command );
102             }
103         }
104         report();
105         atm.println( "Goodbye from " + bankName );
106     }
107
108
109     // Open a new bank account,
110     // prompting the user for information.
111
112     private void openNewAccount()
```

```
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("    transfered "
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
215      // Prompt for an account name (or number), look it up
216      // in the account list. If it's there, return it;
217      // otherwise report an error and return null.
218
219      private BankAccount whichAccount()
220      {
221          String accountName = atm.readWord( "account name: " );
222          BankAccount account = (BankAccount) accountList.get(accountName);
223          if (account == null) {
224              atm.println( "not a valid account" );
```

```
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 exception) {
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
275      // Welcome the user to the bank and instruct her on
276      //  her options.
277
278      private void instructUser()
279      {
280          atm.println( "Welcome to " + bankName );
```



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



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

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

```
 1  // joi/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 performe
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
33      protected BankAccount( int initialBalance, Bank issuingBank )
34          throws InsufficientFundsException
35      {
36          this.issuingBank = issuingBank;
37          deposit( initialBalance );
38      }
39
40      /**
41       * Get transaction fee.  By default, 0.
42       * Override this for accounts having transaction fees.
43       *
44       * @return the fee.
45       */
46
47      protected int getTransactionFee()
48      {
49          return 0;
50      }
51
52      /**
53       * The bank that issued this account.
54       *
55       * @return the Bank.
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
124      public int getBalance()
125      {
126          return balance;
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       *
136       * @exception InsufficientFundsException when appropriate.
137       */
138
139      public final void incrementBalance( int amount )
140          throws InsufficientFundsException
141      {
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          throws InsufficientFundsException
173      {
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          throws InsufficientFundsException;
187  }
```

```
// joi/7/bank/CheckingAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A CheckingAccount is a BankAccount with one new feature:
 * the ability to cash a check by calling the honorCheck method.
 * Each honored check costs the customer a checkFee.
 *
 * @see BankAccount
 *
 * @version 7
 */

public class CheckingAccount extends BankAccount
{
    /**
     * Constructs a CheckingAccount with the given
     * initial balance and issuing Bank.
     * Counts as this account's first transaction.
     *
     * @param initialBalance the opening balance for this account.
     * @param issuingBank the bank that issued this account.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public CheckingAccount( int initialBalance, Bank issuingBank )
        throws InsufficientFundsException
    {
        super( initialBalance, issuingBank );
    }

    /**
     * Honor a check:
     * Charge the account the appropriate fee
     * and withdraw the amount.
     *
     * @param amount amount (in whole dollars) to be withdrawn.
     * @return the amount withdrawn.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public int honorCheck( int amount )
        throws InsufficientFundsException
    {
        // careful error checking logic:
        // first try to deduct the check fee
        // if you succeed, try to honor check
        // if that fails, remember to add back the check fee!

        try {
            incrementBalance( - getIssuingBank().getCheckFee() );
        }
```



```
        catch (InsufficientFundsException e) {
            throw new InsufficientFundsException(
                        "to cover check fee" );
        }
        try {
            withdraw( amount );
        }
        catch (InsufficientFundsException e) {
            incrementBalance( getIssuingBank().getCheckFee() );
            throw new InsufficientFundsException(
                        "to cover check + check fee" );
        }
        return amount;
    }

    /**
     *  Nothing special happens to a CheckingAccount on the
     *  first day of the month.
     */

    public void newMonth()
    {
        return;
    }
}
```

```
// joi/7/bank/SavingsAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A SavingsAccount is a BankAccount that bears interest.
 * A fee is charged for too many transactions in a month.
 *
 * @see BankAccount
 *
 * @version 7
 */

public class SavingsAccount extends BankAccount
{
    private int transactionsThisMonth;

    /**
     * Override getTransactionFee() to return a non-zero fee
     * after the appropriate number of free monthly transactions.
     *
     * @return the fee for current transaction.
     */

    protected int getTransactionFee()
    {
        if (transactionsThisMonth >
            getIssuingBank().getMaxFreeTransactions()) {
            return getIssuingBank().getTransactionFee();
        }
        else {
            return 0;
        }
    }

    /**
     * Increment count of transactions, for this account for
     * this Month and in total  and for the issuing Bank, by one.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public void countTransaction()
        throws InsufficientFundsException
    {
        transactionsThisMonth++;
        super.countTransaction();
    }

    /**
     * Constructor, accepting an initial balance.
     * @param initialBalance the opening balance.
     *
     * @param issuingBank the bank that issued this account.
     *
```



```
     * @exception InsufficientFundsException when appropriate.
     */

    public SavingsAccount( int initialBalance, Bank issuingBank )
         throws InsufficientFundsException
    {
        super( initialBalance, issuingBank);
        transactionsThisMonth = 1;
    }

    /**
     * A SavingsAccount earns interest each month.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public void newMonth()
         throws InsufficientFundsException
    {
        double monthlyRate = getIssuingBank().getInterestRate()/12;
        incrementBalance( (int)(monthlyRate * getBalance()));
        transactionsThisMonth = 0;
    }
}
```

```
// joi/7/bank/FeeAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A FeeAccount is a BankAccount with one new feature:
 * the user is charged for each transaction.
 *
 * @see BankAccount
 *
 * @version 7
 */

public class FeeAccount extends BankAccount
{
    /**
     * Constructor, accepting an initial balance and issuing Bank.
     *
     * @param initialBalance the opening balance.
     * @param issuingBank the bank that issued this account.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public FeeAccount( int initialBalance, Bank issuingBank )
        throws InsufficientFundsException
    {
        super( initialBalance, issuingBank);
    }

    /**
     * The Bank's transaction fee.
     *
     * @return the fee.
     */

    protected int getTransactionFee()
    {
        return getIssuingBank().getTransactionFee();
    }

    /**
     * The way a transaction is counted for a FeeAccount: it levies
     * a transaction fee as well as counting the transaction.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public void countTransaction()
        throws InsufficientFundsException
    {
        incrementBalance( - getTransactionFee() );
        super.countTransaction();
    }

```



```
    /**
     * A FeeAccount incurs a monthly charge.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public void newMonth()
         throws InsufficientFundsException
    {
        incrementBalance( - getIssuingBank().getMonthlyCharge());
    }
}
```

```
 1  // joi/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  }
```

```
// joi/7/bank/class Month
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

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

/**
 * The Month class implements an object that keeps
 * track of the month of the year.
 *
 * @version 7
 */

public class Month
{
    private static final String[] monthName =
        {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
         "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

    private int month;
    private int year;

    /**
     *  Month constructor constructs a Month object
     *  initialized to the current month and year.
     */

    public Month()
    {
        Calendar rightNow = Calendar.getInstance();
        month = rightNow.get( Calendar.MONTH );
        year  = rightNow.get( Calendar.YEAR );
    }

    /**
     *  Advance to next month.
     */

    public void next()
    {
        month = (month + 1) % 12;
        if (month == 0) {
            year++;
        }
    }

    /**
     * How a Month is displayed as a String -
     * for example, "Jan, 2003".
     *
     * @return String representation of the month.
     */

    public String toString()
```



```
    {
        return monthName[month] + ", " + year;
    }

    /**
     * For unit testing.
     */

    public static void main( String[] args )
    {
        Month m = new Month();
        for (int i=0; i < 14; i++, m.next()) {
            System.out.println(m);
        }
        for (int i=0; i < 35; i++, m.next()); // no loop body
        System.out.println( "three years later: " + m );
        for (int i=0; i < 120; i++, m.next());// no loop body
        System.out.println( "ten years later: " + m );
    }
}
```

```
 1  // joi/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  // joi/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  }
```

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

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

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

public class Juno
{
    private final static String os      = "Juno";
    private final static String version = "7";

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

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

    private ShellCommandTable commandTable; // shell commands

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

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

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

        // the file system

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



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

        // create, then start a command line login interpreter

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

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

    public String getHostName()
    {
        return hostName;
    }

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

    public String getOS()
    {
        return os;
    }

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

    public String getVersion()
    {
        return version;
    }

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

    public Directory getUserHomes()
    {
        return userHomes;
    }
```

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

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

import java.util.*;

/**
 * Interpreter for Juno login commands.
 *
 * There are so few commands that if-then-else logic is OK.
 *
 * @version 7
 */

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

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

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

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

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

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

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

    public void CLILogin()
    {
```



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

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

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

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

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

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

```

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

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

import java.util.*;

/**
 * Models a shell (command interpreter)
 *
 * The Shell knows the (Juno) system it's working in,
 * the User who started it,
 * and the console to which to send output.
 *
 * It keeps track of the the current working directory (.) .
 *
 * @version 7
 */

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

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

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

    // Run the command line interpreter

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

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



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

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

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

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

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

    public String getPrompt()
    {
```

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



```
169       }
170   }
```

```
 1  // joi/7/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 7
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       * @exception JunoException for reporting errors
54       */
55
56      public abstract void doIt( StringTokenizer args, Shell sh )
```



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

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

import java.util.*;

/**
 * A ShellCommandTable object maintains a dispatch table of
 * ShellCommand objects keyed by the command names used to invoke
 * them.
 *
 * To add a new shell command to the table, install it from
 * method fillTable().
 *
 * @see ShellCommand
 *
 * @version 7
 */

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

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

    public ShellCommandTable()
    {
        fillTable();
    }

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

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

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



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

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

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

    // Associate a command name with a ShellCommand.

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

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

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

```
 1  // joi/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           throws JunoException
37      {
38          String filename = args.nextToken();
39          new Directory( filename, sh.getUser(), sh.getDot() );
40      }
41  }
```

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

import java.util.*;

/**
 * The Juno shell command to display the contents of a
 * text file.
 * Usage:
 * <pre>
 *     type textfile
 * </pre>
 *
 * @version 7
 */

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

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

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

```
 1  // joi/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           throws JunoException
36      {
37          // Get command keys from global table, print them out.
38
39          sh.getConsole().println( "shell commands" );
40          ShellCommandTable table = sh.getSystem().getCommandTable();
41          String[] names = table.getCommandNames();
42          for (int i = 0; i < names.length; i++ ) {
43              String cmdname = names[i];
44              ShellCommand cmd =
45                  (ShellCommand) table.lookup( cmdname );
46              sh.getConsole().
47                  println( "  " + cmdname + ": " + cmd.getHelpString() );
48          }
49      }
50  }
```

```
 1  // joi/7/juno/NewfileCommand.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 text file.
10   * Usage:
11   * <pre>
12   *     newfile filename contents
13   * </pre>
14   *
15   * @version 7
16   */
17
18  public class NewfileCommand extends ShellCommand
19  {
20      NewfileCommand()
21      {
22          super( "create a new TextFile", "filename contents" );
23      }
24
25      /**
26       * Create a new TextFile in the current Directory.
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;
38          String contents;
39          filename = args.nextToken();
40          contents  = args.nextToken("").trim(); // rest of line
41          new TextFile( filename, sh.getUser(),
42                        sh.getDot(), contents );
43      }
44  }
```

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

import java.util.*;

/**
 * The Juno shell command to change directory.
 * Usage:
 * <pre>
 *     cd [directory]
 * </pre>
 * for moving to the named directory.
 *
 * @version 7
 */

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

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

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



```
}
```

```
 1  // joi/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  // joi/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           throws JunoException
36      {
37          throw new ExitShellException();
38      }
39  }
```

```
 1  // joi/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           throws JunoException
36      {
37          String filename = args.nextToken();
38          sh.getDot().removeJFile(filename);
39      }
40  }
```

```
// joi/7/jfiles/JFile.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

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

/**
 * A JFile object models a file in a hierarchical file system.
 * <p>
 * Extend this abstract class to create particular kinds of JFiles,
 * e.g.:<br>
 * Directory  -
 *    a JFile that maintains a list of the files it contains.<br>
 * TextFile  -
 *    a JFile containing text you might want to read.<br>
 *
 * @see Directory
 * @see TextFile
 *
 * @version 7
 */

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

    public static final String separator = File.separator;

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

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

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



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

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

    public String getName()
    {
        return name;
    }

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

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

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

    public abstract int getSize();

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

    public abstract String getSuffix();

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

    public void setOwner( User owner )
```

```
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  }
```

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

import java.util.*;

/**
 * Directory of JFiles.
 *
 * A Directory is a JFile that maintains a
 * table of the JFiles it contains.
 *
 * @version 7
 */

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

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

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

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

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

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

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



```
    }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    /**
```



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

    public String getSuffix()
    {
        return "";
    }

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

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

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

    public String getContents()
    {
        return contents;
    }

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

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


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

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

```
 1  // joi/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  }
```

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

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

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

    public JunoException()
    {
    }

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

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

        super( message );
    }

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

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

```
 1  // joi/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  }
```

```
// joi/8/terminal/Terminal.java
// (and terminal/Terminal.java)
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.io.*;

/**
 * Terminal provides a user-friendly interface to the standard System
 * input and output streams (in, out, and err).
 * <p>
 * A Terminal is an object.  In general, one is expected to instantiate
 * just one Terminal.  Although one might instantiate several, all will
 * share the same System streams.
 * <p>
 * A Terminal may either explicitly echo input, or not.  Echoing input
 * is useful, for example, when testing with I/O redirection.
 * <p>
 * Inspired by Cay Horstmann's Console Class.
 */

public class Terminal
{
    private boolean echo = false;
    private static BufferedReader in =
        new BufferedReader(new FileReader(FileDescriptor.in));


    // Print a prompt to the console without a newline.

    private void printPrompt( String prompt )
    {
        print( prompt );
        System.out.flush();
    }

    /**
     * Construct a Terminal that doesn't echo input.
     */

    public Terminal()
    {
        this( false );
    }

    /**
     * Construct a Terminal.
     *
     * @param echo whether or not input should be echoed.
     */

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

```



```
    /**
     * Read a line (terminated by a newline) from the Terminal.
     * @param prompt output string to prompt for input.
     *
     * @return the string (without the newline character),
     *  null if eof.
     */

    public String readLine( String prompt )
    {
        printPrompt(prompt);
        try {
            String line = in.readLine();
            if (echo) {
                println(line);
            }
            return line;
        }
        catch (IOException e) {
            return null;
        }
    }

    /**
     * Read a line (terminated by a newline) from the Terminal.
     *
     * @return the string (without the newline character).
     */

    public String readLine()
    {
        return readLine( "" );
    }

    // Read a line from the Terminal.  An end of file,
    // indicated by a null, raises a runtime exception.
    // Used only internally.

    private String readNonNullLine()
    {
        return readNonNullLine( "" );
    }

    // Read a line from the Terminal.  An end of file,
    // indicated by a null, raises a runtime exception.
    // Used only internally.

    private String readNonNullLine( String prompt )
    {
        String line = readLine( prompt );
        if (line == null ) {
            throw new RuntimeException( "End of File encountered." );
        }
        return line;
    }

```

```
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       *
122       * @return the word read.
123       */
124
125      public String readWord( String prompt )
126      {
127          String line = readNonNullLine( prompt );
128          if (line.length() == 0) {
129              println( "Empty line.  Please try again." );
130              return readWord("");
131          }
132          line = line.trim();
133          for ( int i = 0; i < line.length(); i++ ) {
134              if ( Character.isWhitespace( line.charAt(i) ) ) {
135                  return line.substring( 0, i );
136              }
137          }
138          return line;
139      }
140
141      /**
142       * Read a word from the Terminal.
143       * If an empty line is entered, try again.
144       * Words are terminated by whitespace.
145       * Leading whitespace is trimmed; the rest of the line
146       * is disposed of.
147       *
148       * @return the word read.
149       */
150
151      public String readWord()
152      {
153          return readWord( "" );
154      }
155
156      /**
157       * Read a word from the Terminal.
158       * If an empty line is entered, throw an exception.
159       * Words are terminated by whitespace.
160       * Leading whitespace is trimmed; the rest of the line
161       * is disposed of.
162       *
163       * @param prompt output string to prompt for input.
164       * @return the word read.
165       *
166       * @throws RuntimeException if it reads an empty line.
167       */
168
```



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

```
225       * @return the character read.
226       *
227       * @throws RuntimeException if it reads an empty line.
228       */
229
230      public char readCharOnce( String prompt )
231      {
232          String line = readNonNullLine(prompt);
233          if (line.length() == 0) {
234              throw new RuntimeException("Empty line encountered.");
235          }
236          return line.charAt(0);
237      }
238
239      /**
240       * Read a character from the Terminal.
241       * Prompt again when an empty line is read.
242       *
243       * @param prompt output string to prompt for input.
244       *
245       * @return the character read.
246       */
247
248      public char readChar()
249      {
250          return readChar("");
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
262      public char readCharOnce()
263      {
264          return readCharOnce("");
265      }
266
267      /**
268       * Read "yes" or "no" from the Terminal.
269       * If an empty line or improper character is read,
270       * try again.
271       * Look only at first character and accept any case.
272       *
273       * @param prompt output string to prompt for input.
274       * @return true if yes, false if no.
275       */
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       * Read "yes" or "no" from the Terminal.
296       * If an empty line or improper character is read,
297       * throw an exception.
298       * Look only at first character and accept any case.
299       *
300       * @param prompt output string to prompt for input.
301       * @return true if yes, false if no.
302       *
303       * @throws RuntimeException on improper input.
304       */
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       * Read "yes" or "no" from the Terminal.
325       * If an empty line or improper character is read,
326       * try again. No prompting is done.
327       * Look only at first character and accept any case.
328       *
329       * @return true if yes, false if no.
330       */
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      * @param prompt output string to prompt for input.
395      * @return the input value as an int.
396      *
397      * @throws NumberFormatException for a badly formed integer.
398      */
399
400     public int readIntOnce( String prompt )
401         throws NumberFormatException
402     {
403         return Integer.parseInt(readNonNullLine( prompt ).trim());
404     }
405
406     /**
407      * Read an integer, terminated by a new line, from the Terminal.
408      * If a NumberFormatException is encountered, try again.
409      *
410      * @return the input value as an int.
411      */
412
413     public int readInt()
414     {
415         return readInt("");
416     }
417
418     /**
419      * Read an integer, terminated by a new line, from the Terminal.
420      *
421      * @return the input value as an int.
422      *
423      * @throws NumberFormatException for a badly formed integer.
424      */
425
426     public int readIntOnce()
427         throws NumberFormatException
428     {
429         return readIntOnce("");
430     }
431
432     /**
433      * Read a double-precision floating point number,
434      * terminated by a newline, from the Terminal.
435      * If a NumberFormatException is encountered, try again.
436      *
437      * @param prompt output string to prompt for input.
438      * @return the input value as a double.
439      */
440
441     public double readDouble( String prompt )
442     {
443         while( true ) {
444             try {
445                 return Double.
446                     parseDouble(readNonNullLine( prompt ).trim());
447             }
448             catch (NumberFormatException e) {
```

```
println("Not a floating point number. Please try again.");
            }
        }
    }

    /**
     * Read a double-precision floating point number,
     * terminated by a newline, from the Terminal.
     *
     * @param prompt output string to prompt for input.
     * @return the input value as a double.
     *
     * @throws NumberFormatException for a badly formed number.
     */

    public double readDoubleOnce( String prompt )
        throws NumberFormatException
    {
        return Double.parseDouble(readNonNullLine( prompt ).trim());
    }

    /**
     * Read a double-precision floating point number,
     * terminated by a newline, from the Terminal.
     * If a NumberFormatException is encountered, try again.
     *
     * @return the input value as a double.
     */

    public double readDouble()
    {
        return readDouble("");
    }

    /**
     * Read a double-precision floating point number,
     * terminated by a newline, from the Terminal.
     *
     * @return the input value as a double.
     *
     * @throws NumberFormatException for a badly formed number.
     */

    public double readDoubleOnce()
        throws NumberFormatException
    {
        return readDouble("");
    }

    /**
     * Print a Boolean value
     * (<code>true</code> or <code>false</code>)
     * to standard output (without a newline).
     *
     * @param b Boolean to print.
     */
```



```

    public void print( boolean b )
    {
        System.out.print( b );
    }

    /**
     * Print character to standard output (without a newline).
     *
     * @param ch character to print.
     */

    public void print( char ch )
    {
        System.out.print( ch );
    }

    /**
     * Print character array to standard output (without a newline).
     *
     * @param s character array to print.
     */

    public void print( char[] s )
    {
        System.out.print( s );
    }

    /**
     * Print a double-precision floating point number to standard
     * output (without a newline).
     *
     * @param val number to print.
     */

    public void print( double val )
    {
        System.out.print( val );
    }

    /**
     * Print a floating point number to standard output
     * (without a newline).
     *
     * @param val number to print.
     */

    public void print( float val )
    {
        System.out.print( val );
    }

    /**
     * Print integer to standard output (without a newline).
     *
     * @param val integer to print.
```

```
561       */
562
563      public void print( int val )
564      {
565          System.out.print( val );
566      }
567
568      /**
569       * Print a long integer to standard output (without a newline).
570       *
571       * @param val integer to print.
572       */
573
574      public void print( long val )
575      {
576          System.out.print( val );
577      }
578
579      /**
580       * Print Object to standard output (without a newline).
581       *
582       * @param val Object to print.
583       */
584
585      public void print( Object val )
586      {
587          System.out.print( val.toString() );
588      }
589
590      /**
591       * Print string to standard output (without a newline).
592       *
593       * @param str String to print.
594       */
595
596      public void print( String str )
597      {
598          System.out.print( str );
599      }
600
601      /**
602       * Print a newline to standard output,
603       * terminating the current line.
604       */
605
606      public void println()
607      {
608          System.out.println();
609      }
610
611      /**
612       * Print a Boolean value
613       * (<code>true</code> or <code>false</code>)
614       * to standard output, followed by a newline.
615       * @param b Boolean to print.
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
676      public void println( int val )
677      {
678          System.out.println( val );
679      }
680
681      /**
682       * Print a long integer to standard output,
683       * followed by a newline.
684       *
685       * @param val long integer to print.
686       */
687
688      public void println( long val )
689      {
690          System.out.println( val );
691      }
692
693      /**
694       * Print Object to standard output, followed by a newline.
695       *
696       * @param val Object to print
697       */
698
699      public void println( Object val )
700      {
701          System.out.println( val.toString() );
702      }
703
704      /**
705       * Print string to standard output, followed by a newline.
706       *
707       * @param str String to print
708       */
709
710      public void println( String str )
711      {
712          System.out.println( str );
713      }
714
715      /**
716       * Print a Boolean value
717       * (<code>true</code> or <code>false</code>)
718       * to standard err (without a newline).
719       *
720       * @param b Boolean to print.
721       */
722
723      public void errPrint( boolean b )
724      {
725          System.err.print( b );
726      }
727
728      /**
```



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

```
    /**
     * Print a long integer to standard err (without a newline).
     *
     * @param val integer to print.
     */

    public void errPrint( long val )
    {
        System.err.print( val );
    }

    /**
     * Print Object to standard err (without a newline).
     *
     * @param val Object to print.
     */

    public void errPrint( Object val )
    {
        System.err.print( val.toString() );
    }

    /**
     * Print string to standard err (without a newline).
     *
     * @param str String to print.
     */

    public void errPrint( String str )
    {
        System.err.print( str );
    }

    /**
     * Print a newline to standard err,
     * terminating the current line.
     */

    public void errPrintln()
    {
        System.err.println();
    }

    /**
     * Print a Boolean value
     * (<code>true</code> or <code>false</code>)
     * to standard err, followed by a newline.
     *
     * @param b Boolean to print.
     */

    public void errPrintln( boolean b )
    {
        System.err.println( b );
    }

```



```
    /**
     * Print character to standard err, followed by a newline.
     *
     * @param ch character to print.
     */

    public void errPrintln( char ch )
    {
        System.err.println( ch );
    }

    /**
     * Print a character array to standard err,
     * followed by a newline.
     *
     * @param s character array to print.
     */

    public void errPrintln( char[] s )
    {
        System.err.println( s );
    }

    /**
     * Print floating point number to standard err,
     * followed by a newline.
     *
     * @param val number to print.
     */

    public void errPrintln( float val )
    {
        System.err.println( val );
    }

    /**
     * Print a double-precision floating point number to
     * standard err, followed by a newline.
     *
     * @param val number to print.
     */

    public void errPrintln( double val )
    {
        System.err.println( val );
    }

    /**
     * Print integer to standard err, followed by a newline.
     *
     * @param val integer to print.
     */

    public void errPrintln( int val )
    {
        System.err.println( val );
```

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



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

```
// joi/8/juno/Password.java//
//
//  Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * Model a good password.
 *
 * <p>
 * A password is a String satisfying the following conditions
 * (close to those required of Unix passwords, according to
 * the <code> man passwd </code> command in Unix):
 * <br>
 * <ul>
 * <li> A password must have at least PASSLENGTH characters, where
 *      PASSLENGTH defaults to 6. Only the first eight characters
 *      are significant.
 *
 * <li> A password must contain at least two alphabetic characters
 *      and at least one numeric or special character. In this  case,
 *      "alphabetic" refers to all upper or lower case letters.
 *
 * <li> A password must not contain a specified string as a substring
 *      For comparison purposes, an upper case letter and its
 *      corresponding lower case letter are equivalent.
 *
 * <li> A password must not be a substring of a specified string.
 *      For comparison purposes, an upper case letter and its
 *      corresponding lower case letter are equivalent.
 *
 * </ul>
 * <br>
 * A password string may be stored in a Password object only in
 * encrypted form.
 */

public class Password
{
    private String password;

    /**
     * Construct a new Password.
     *
     * @param password the new password.
     * @param notSubstringOf a String that may not contain the password.
     * @param doesNotContain a String the password may not contain.
     *
     * @exception BadPasswordException when password is unacceptable.
     */

    public Password(String password, String notSubstringOf,
                    String doesNotContain)
         throws BadPasswordException
    {
        // if password is not acceptable
        // throw new BadPasswordException( reason )
```



```
        this.password = encrypt(password);
    }

    // Rewrite s in a form that makes it hard to guess s.

    private String encrypt( String s )
    {
        return Integer.toHexString(s.hashCode());
    }

    /**
     * See whether a supplied guess matches this password.
     *
     * @param guess the trial password.
     *
     * @exception BadPasswordException when match fails.
     */

    public void match(String guess)
         throws BadPasswordException
    {
    }

    /**
     * Unit test for Password objects.
     */

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

```
 1  // joi/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  // joi/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                  System.err.println("Unable to close input stream.");
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  // joi/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  }
```

```
// joi/9/bank/Bank.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

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

/**
 * A Bank object simulates the behavior of a simple bank/ATM.
 * It contains a Terminal object and a collection of
 * BankAccount objects.
 *
 * The visit method opens this Bank for business,
 * prompting the customer for input.
 *
 * It is persistent: it can save its state to a file and read it
 * back at a later time.
 *
 * To create a Bank and open it for business issue the command
 * <code>java Bank</code> with appropriate arguments.
 *
 * @see BankAccount
 * @version 9
 */

public class Bank
    implements Serializable
{
    private String bankName;           // the name of this Bank
    private transient Terminal atm;   // for communication with world
    private int balance = 0;           // total cash on hand
    private int transactionCount = 0;  // number of Bank transactions
    private Month month;               // the current month.
    private Map accountList;           // mapping names to accounts.

    private int checkFee = 2;            // cost for each check
    private int transactionFee = 1;      // fee for each transaction
    private int monthlyCharge = 5;       // monthly charge
    private double interestRate = 0.05;  // annual rate paid on savings
    private int maxFreeTransactions = 3; // for savings accounts

    // what the banker can ask of the bank

    private static final String BANKER_COMMANDS =
    "Banker commands: " +
        "exit, open, customer, nextmonth, report, help.";

    // what the customer can ask of the bank

    private static final String CUSTOMER_TRANSACTIONS =
    "    Customer transactions: deposit, withdraw, transfer,\n" +
    "     balance, cash check, quit, help.";

    /**
     * Construct a Bank with the given name.
```



```
     *
     * @param bankName the name for this Bank.
     */

    public Bank( String bankName )
    {
        this.atm      = atm;
        this.bankName = bankName;
        accountList   = new TreeMap();
        month         = new Month();
    }

    /**
     * Simulates interaction with a Bank.
     * Presents the user with an interactive loop, prompting for
     * banker transactions and in the case of the banker
     * transaction "customer", an account id and further
     * customer transactions.
     */

    public void visit()
    {
        instructUser();

        String command;
        while (!(command =
                 atm.readWord("banker command: ")).equals("exit")) {

            if (command.startsWith("h")) {
                help( BANKER_COMMANDS );
            }
            else if (command.startsWith("o")) {
                openNewAccount();
            }
            else if (command.startsWith("n")) {
                newMonth();
            }
            else if (command.startsWith("r")) {
                report();
            }
            else if (command.startsWith( "c" ) ) {
                BankAccount acct = whichAccount();
                if ( acct != null ) {
                        processTransactionsForAccount( acct );
                }
            }
            else {
                // Unrecognized Request
                atm.println( "unknown command: " + command );
            }
        }
        report();
        atm.println( "Goodbye from " + bankName );
    }
```

```
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("    transfered "
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
219      // Prompt for an account name (or number), look it up
220      // in the account list. If it's there, return it;
221      // otherwise report an error and return null.
222
223      private BankAccount whichAccount()
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 exception) {
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
279      // Welcome the user to the bank and instruct her on
280      //  her options.
```



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



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

```
449       *         </pre>
450       */
451
452      public static void main( String[] args )
453      {
454          boolean echo        = false;
455          String bankFileName = 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                  bankFileName = args[++i];
467                  continue;
468              }
469          }
470
471          // create a new Bank or read one from a file
472          if (bankFileName == null) {
473              theBank = new Bank( bankName );
474          }
475          else {
476              theBank = readBank( bankName, bankFileName );
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 (bankFileName != null) {
485              writeBank(theBank, bankFileName);
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 bankFileName the name of the file containing the Bank
493      //
494      // @return the Bank
495
496      private static Bank readBank(String bankName, String bankFileName)
497      {
498          File file = new File( bankFileName );
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 " + bankFileName);
509              return bank;
510          }
511          catch (Exception e ) {
512              System.err.println(
513                  "Problem reading " + bankFileName );
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
528      // Write a Bank to a file.
529      //
530      // @param bank     the Bank
531      // @param fileName the name of the file to write the Bank to
532
533      private static void writeBank( Bank bank, String fileName)
534      {
535          ObjectOutputStream outStream = null;
536          try {
537              outStream = new ObjectOutputStream(
538                              new FileOutputStream( fileName ) );
539              outStream.writeObject( bank );
540              System.out.println(
541                 "Bank state written to file " + fileName);
542          }
543          catch (Exception e ) {
544              System.err.println(
545                 "Problem writing " + fileName );
546          }
547          finally {
548              try {
549                  outStream.close();
550              }
551              catch (Exception e ) {
552              }
553          }
554      }
555  }
```

```
// joi/9/bank/BankAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.io.Serializable;

/**
 * A BankAccount object has private fields to keep track
 * of its current balance, the number of transactions
 * performed and the Bank in which it is an account, and
 * and public methods to access those fields appropriately.
 *
 * @see Bank
 * @version 9
 */

public abstract class BankAccount
    implements Serializable
{
    private int  balance = 0;          // Account balance (whole dollars)
    private int  transactionCount = 0; // Number of transactions performe
    private Bank issuingBank;          // Bank issuing this account

    /**
     * Construct a BankAccount with the given initial balance and
     * issuing Bank. Construction counts as this BankAccount's
     * first transaction.
     *
     * @param initialBalance the opening balance.
     * @param issuingBank the bank that issued this account.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    protected BankAccount( int initialBalance, Bank issuingBank )
        throws InsufficientFundsException
    {
        this.issuingBank = issuingBank;
        deposit( initialBalance );
    }

    /**
     * Get transaction fee.  By default, 0.
     * Override this for accounts having transaction fees.
     *
     * @return the fee.
     */

    protected int getTransactionFee()
    {
        return 0;
    }

    /**
     * The bank that issued this account.
```



```
     *
     * @return the Bank.
     */

    protected Bank getIssuingBank()
    {
        return issuingBank;
    }

    /**
     * Withdraw the given amount, decreasing this BankAccount's
     * balance and the issuing Bank's balance.
     * Counts as a transaction.
     *
     * @param amount the amount to be withdrawn
     * @return amount withdrawn
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public int withdraw( int amount )
        throws InsufficientFundsException
    {
        incrementBalance( -amount - getTransactionFee() );
        countTransaction();
        return amount ;
    }

    /**
     * Deposit the given amount, increasing this BankAccount's
     * balance and the issuing Bank's balance.
     * Counts as a transaction.
     *
     * @param amount the amount to be deposited
     * @return amount deposited
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public int deposit(int amount)
        throws InsufficientFundsException
    {
        incrementBalance( amount - getTransactionFee() );
        countTransaction();
        return amount ;
    }

    /**
     * Request for balance. Counts as a transaction.
     *
     * @return current account balance.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public int requestBalance()
```

```
113          throws InsufficientFundsException
114      {
115          incrementBalance( - getTransactionFee() );
116          countTransaction();
117          return getBalance() ;
118      }
119
120      /**
121       * Get the current balance.
122       * Does NOT count as a transaction.
123       *
124       * @return current account balance
125       */
126
127      public int getBalance()
128      {
129          return balance;
130      }
131
132      /**
133       * Increment account balance by given amount.
134       * Also increment issuing Bank's balance.
135       * Does NOT count as a transaction.
136       *
137       * @param amount the amount of the increment.
138       *
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       * Get the number of transactions performed by this
156       * account. Does NOT count as a transaction.
157       *
158       * @return number of transactions performed.
159       */
160
161      public int getTransactionCount()
162      {
163          return transactionCount;
164      }
165
166      /**
167       * Increment by 1 the count of transactions, for this account
168       * and for the issuing Bank.
```



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

```
// joi/9/bank/class Month
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

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

/**
 * The Month class implements an object that keeps
 * track of the month of the year.
 *
 * @version 9
 */

public class Month
    implements Serializable
{
    private static final String[] monthName =
        {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
         "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

    private int month;
    private int year;

    /**
     *  Month constructor constructs a Month object
     *  initialized to the current month and year.
     */

    public Month()
    {
        Calendar rightNow = Calendar.getInstance();
        month = rightNow.get( Calendar.MONTH );
        year  = rightNow.get( Calendar.YEAR );
    }

    /**
     *  Advance to next month.
     */

    public void next()
    {
        month = (month + 1) % 12;
        if (month == 0) {
            year++;
        }
    }

    /**
     * How a Month is displayed as a String -
     * for example, "Jan, 2003".
     *
     * @return String representation of the month.
     */

```



```
    public String toString()
    {
        return monthName[month] + ", " + year;
    }

    /**
     * For unit testing.
     */

    public static void main( String[] args )
    {
        Month m = new Month();
        for (int i=0; i < 14; i++, m.next()) {
            System.out.println(m);
        }
        for (int i=0; i < 35; i++, m.next()); // no loop body
        System.out.println( "three years later: " + m );
        for (int i=0; i < 120; i++, m.next());// no loop body
        System.out.println( "ten years later: " + m );
    }
}
```

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

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/**
 * A JOIPanel displays a button and a message.
 * Pushing the button changes the message.
 *
 * This panel can be displayed either from an applet
 * in a browser or by the JVM as an application.
 *
 * @version 10
 */

public class JOIPanel extends Applet
{
    private static final String MESSAGE1 = "Java Outside In";
    private static final String MESSAGE2 = "Java Inside Out";
    private String currentMessage = MESSAGE1; // currently displayed

    private Font font;                // for printing the message
    private Button button;            // for changing messages

    /**
     * Equip this Panel with a Button
     * and an associated ButtonListener, and
     * set the font for the message.
     */

    public void init()
    {
        // what this Panel looks like
        button = new Button( "Press Me" );
        this.add( button );
        font = new Font("Garamond", Font.BOLD, 48);

        // how this Panel behaves
        button.addActionListener( new JOIButtonListener( this ) );
    }

    /**
     * Method that responds when the ButtonListener sends a
     * changeMessage message.
     */

    public void changeMessage()
    {
        currentMessage =
            currentMessage.equals(MESSAGE1) ? MESSAGE2 : MESSAGE1;
        this.repaint();
    }
```



```

    /**
     * Draw the current message on this Panel.
     *
     * (The button is already there.)
     *
     * @param g an object encapsulating the graphics (e.g. pen)
     *          properties.
     */

    public void paint(Graphics g)
    {
        g.setColor(Color.black);
        g.setFont(font);
        g.drawString(currentMessage, 40, 75);
    }

    /**
     * Ask the JVM to display this Panel.
     */

    public static void main( String[] args )
    {
        Terminal t     = new Terminal();
        Frame frame    = new Frame();
        JOIPanel panel = new JOIPanel();
        panel.init();
        frame.add(panel);
        frame.setSize(400,120);
        frame.show();
        t.readLine("Type return to close the window ... ");
        System.exit(0);
    }
}
```

```
 1  // joi/10/joi/JOIButtonListener.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/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 JOIPanel 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 10
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 ButtonListener, 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>
```

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

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

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

public class Juno
    implements Serializable
{
    private final static String os      = "Juno";
    private final static String version = "10";

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

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

    private ShellCommandTable commandTable; // shell commands

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

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

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

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



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

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

        // the file system

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

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

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

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

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

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

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

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

    public void shutDown( )
```

```
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];
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 outStream = null;
326          try {
327              outStream = new ObjectOutputStream(
328                                     new FileOutputStream( fileName ) );
329              outStream.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                  outStream.close();
340              }
341              catch (Exception e ) {
342              }
343          }
344      }
345  }
```

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

import java.util.*;

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

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

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

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

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

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

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

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

    public void CLILogin()
    {
```



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


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

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

                // NullPointerException if no such user,
                // JunoException if password fails to match -
                // message to user doesn't give away which.
```

```
                // The sysadmin would probably want a log
                // that did keep track.
                //
                // Other exceptions should be caught in shell()

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

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

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

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

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

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

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



```

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

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

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


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

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

    // Remind user of available commands.

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

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

import java.util.*;

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

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

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

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

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

    // Run the command line interpreter

    private void CLIShell()
    {
```



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

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

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

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

```

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



```
169       *
170       * @return the console.
171       */
172
173      public OutputInterface getConsole()
174      {
175          return console;
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  // joi/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  }
```

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

import java.util.*;

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

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

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

    public ShellCommandTable()
    {
        fillTable();
    }

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

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

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



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

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

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

    // Associate a command name with a ShellCommand.

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

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

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

```
 1  // joi/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 10
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           throws JunoException
37      {
38          String filename = args.nextToken();
39          new Directory( filename, sh.getUser(), sh.getDot() );
40      }
41  }
```

```
 1  // joi/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 10
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          try {
40              filename = args.nextToken();
41          }
42          catch (NoSuchElementException e) {
43              throw new BadShellCommandException( this );
44          }
45          try {
46              sh.getConsole().println(
47                  ( (TextFile) sh.getDot().
48                    retrieveJFile( filename ) ).getContents() );
49          }
50          catch (NullPointerException e) {
51              throw new JunoException( "JFile does not exist: "
52                                       + filename);
53          }
54          catch (ClassCastException e) {
55              throw new JunoException( "JFile not a text file: "
56                                       + filename);
```



```
57          }
58      }
59  }
```

```
 1  // joi/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 10
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           throws JunoException
36      {
37          // Get command keys from global table, print them out.
38
39          sh.getConsole().println( "shell commands" );
40          ShellCommandTable table = sh.getSystem().getCommandTable();
41          String[] names = table.getCommandNames();
42          for (int i = 0; i < names.length; i++ ) {
43              String cmdname = names[i];
44              ShellCommand cmd =
45                  (ShellCommand) table.lookup( cmdname );
46              sh.getConsole().
47                  println( "  " + cmdname + ": " + cmd.getHelpString() );
48          }
49      }
50  }
```

```
 1  // joi/10/juno/NewfileCommand.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 text file.
10   * Usage:
11   * <pre>
12   *     newfile filename contents
13   * </pre>
14   *
15   * @version 10
16   */
17
18  public class NewfileCommand extends ShellCommand
19  {
20      NewfileCommand()
21      {
22          super( "create a new TextFile", "filename contents" );
23      }
24
25      /**
26       * Create a new TextFile in the current Directory.
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;
38          String contents;
39          filename = args.nextToken();
40          contents  = args.nextToken("").trim(); // rest of line
41          new TextFile( filename, sh.getUser(),
42                        sh.getDot(), contents );
43      }
44  }
```

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

import java.util.*;

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

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

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

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



```
}
```

```
 1  // joi/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 doIt( 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  }
```

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

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

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

 * <pre>
 *
 * @version 10
 */

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

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

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



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

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

```
 1  // joi/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 10
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().removeJFile(filename);
39      }
40  }
41
```

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

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

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

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

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

    public static final String separator = File.separator;

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

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

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



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

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

    public String getName()
    {
        return name;
    }

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

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

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

    public abstract int getSize();

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

    public abstract String getSuffix();

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

```

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



```
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  }
```

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

import java.util.*;

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

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

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

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

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

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

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

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



```
    }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    /**
```



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

    public String getSuffix()
    {
        return "";
    }

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

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

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

    public String getContents()
    {
        return contents;
    }

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

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


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

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

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

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

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

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

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

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

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

    /**
```



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

    public String getName()
    {
        return name;
    }

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

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

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

    public Directory getHome()
    {
        return home;
    }

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

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

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

```
 1  // joi/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 10
10   */
11
12  class BadShellCommandException extends JunoException
13  {
14      ShellCommand command;
15
16      /**
17       * Construct a new BadShellCommandException
18       * containing the badly used command.
19       *
20       * @param the ShellCommand being misused.
21       */
22
23      public BadShellCommandException( ShellCommand command )
24      {
25          this.command = command;
26      }
27
28      /**
29       * Get the command.
30       */
31
32      public ShellCommand getCommand()
33      {
34          return command;
35      }
36  }
```

```
 1  // joi/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  // joi/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  // joi/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  }
```

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

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

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

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

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

    private String title;  // title for the windows

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

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

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

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

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

        JPanel panel = new JPanel();
```



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

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

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

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

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

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

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

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

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

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

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

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

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



```
169             // Everything will go into a vertical Box, a container
170             // whose contents are laid out using BoxLayout
171
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             box.add( p );
187             box.add( Box.createVerticalStrut( 15 ) );
188
189             // Then a horizontal Box containing the two buttons
190
191             Box row = Box.createHorizontalBox();
192             row.add( Box.createGlue() );
193
194             ok = new JButton( "OK" );
195             row.add( ok );
196             row.add( Box.createGlue() );
197
198             exit = new JButton( "Exit" );
199             row.add( exit );
200             row.add( Box.createGlue() );
201             box.add( row );
202             box.add( Box.createVerticalStrut( 15 ) );
203
204             this.setLayout( new BorderLayout() );
205             this.add( box, BorderLayout.CENTER );
206
207             // Set up the listeners (the semantics)
208
209             ok.addActionListener( new LoginProcessor() );
210             exit.addActionListener( closeMe ); // shuts down Juno
211         }
212
213         // An inner inner class for the semantics
214         // when the user clicks OK.
215
216         private class LoginProcessor implements ActionListener
217         {
218             public void actionPerformed(ActionEvent e)
219             {
220                 String str = nameField.getText() +  " " +
221                     passwordField.getText();
222                 nameField.setText("");
223                 passwordField.setText("");
224                 messages.setText(str+'\n');  // for debugging
```

```
                interpreter.interpret( str );
            }
        }
    }

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

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

        private JButton register;
        private JButton clear;

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

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

            // First a panel containing the text fields

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

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

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

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

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

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

            // Then a horizontal Box containing the buttons

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



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

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

            // Set up the listeners (the semantics)

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

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

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

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

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

```
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
```

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

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

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

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

    // the components on the window

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

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

    private boolean echoInput;

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

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

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

        // set up console's look and feel

```



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

        Box box = Box.createVerticalBox();

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

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

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

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

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

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

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

        // Install menus and tool bar.

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

        JToolBar toolBar  = new JToolBar();

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

```

```
113          ShellCommandTable table = sh.getSystem().getCommandTable();
114          String [] commandNames = table.getCommandNames();
115          for ( int i = 0; i < commandNames.length; i++ ) {
116
117              String commandName = commandNames[i];
118              ShellCommand command =
119                  table.lookup( commandName );
120
121              CommandMenuAction commandAction =
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              button.setToolTipText( command.getHelpString() );
134          }
135
136          this.setJMenuBar( menuBar );
137          this.getContentPane().add( toolBar,
138                                     BorderLayout.NORTH );
139          menuBar.add( commandMenu );
140          menuBar.add( helpMenu );
141
142          pack();
143          show();
144
145          // add Listener to the Do It button
146
147          doIt.addActionListener( new Interpreter() );
148
149          // add listener to the Logout button and window closer
150
151          closeMe = new WindowCloser( this );
152          logout.addActionListener( closeMe );
153          this.addWindowListener( closeMe );
154      }
155
156      // Set the GUI prompt
157
158      private void setPrompt(String prompt)
159      {
160          this.promptLabel.setText(prompt);
161      }
162
163      // Implementing the OutputInterface.
164      // Everything goes to the single message area.
165
166      public void println( String str )
167      {
168          stdout.append(str + "\n");
```



```
169      }
170
171      public void errPrintln( String str )
172      {
173          stderr.append(str + "\n");
174      }
175
176      public boolean isGUI()
177      {
178          return true;
179      }
180
181      public boolean isRemote()
182      {
183          return false;
184      }
185
186      public boolean isEchoInput()
187      {
188          return echoInput;
189      }
190
191      // An inner class for the semantics when the user submits
192      // a ShellCommand for execution.
193
194      private class Interpreter
195          implements ActionListener
196      {
197          public void actionPerformed( ActionEvent e )
198          {
199              String str = commandLine.getText();
200              stdout.append( sh.getPrompt() + str + '\n');
201              if (sh.interpret( str )) {
202                  setPrompt( sh.getPrompt() );
203              }
204              else {
205                  closeMe.actionPerformed(null);
206              }
207          }
208      }
209
210      private class CommandMenuAction extends AbstractAction
211      {
212          private String argString;
213          private String helpString;
214
215          public CommandMenuAction( String text, String argString )
216          {
217              super( text );
218              this. argString = argString;
219          }
220
221          public void actionPerformed( ActionEvent e )
222          {
223              commandLine.setText( getValue( Action.NAME ) +
224                                   " " + argString );
```

```
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  // joi/10/juno/InterpreterInterface.java
 2  //
 3  //
 4  // Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  /**
 7   * Juno 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
25      public boolean interpret( String str );
26  }
```

```
 1  // joi/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
20     public String readLine( String promptString );
21  }
22
```

```
 1  // joi/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
```

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

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

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

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

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

    // Implement InputInterface

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

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

    // Implement OutputInterface

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

    public void println(String str )
```



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

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

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

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

    public boolean isEchoInput()
    {
        return echo;
    }

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

    public boolean isGUI()
    {
        return false;
    }

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

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

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

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

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

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

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

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

    private PrintWriter junoLog;

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

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



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

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

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

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

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

```
113          log("*** end session " + sessionCount);
114      }
115
116      /**
117       * Create the readers and writers for the socket
118       * for this session.
119       */
120
121      private void setUpConnection()
122           throws IOException
123      {
124          in = new BufferedReader(
125                  new InputStreamReader(clientSocket.getInputStream()));
126          out = new PrintWriter(
127                  new OutputStreamWriter(clientSocket.getOutputStream()));
128      }
129
130      // implement the InputInterface
131
132     /**
133      * Read a line (terminated by a newline) from console socket.
134      *
135      * Log the input line before returning it if required.
136      *
137      * @param promptString output string to prompt for input
138      * @return the string (without the newline character)
139      */
140
141      public String readLine( String promptString )
142      {
143          String s = "";
144          this.print(promptString);
145          out.flush();
146          try {
147              s = in.readLine();
148              if (logall) {
149                  log("> " + s);
150              }
151              if (echo) {
152                  out.println(s);
153              }
154          }
155          catch (IOException e) {
156              String msg = "IO error reading from remote console";
157              System.out.println(msg);
158              out.println(msg);
159          }
160          return s;
161      }
162
163      /**
164       * Write a String to console socket.
165       *
166       * Log the output if required.
167       *
168       * @param str - the string to write
```



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

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