```
 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33       * @param amount the amount to be withdrawn.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  }
```

```
// joi/1/bank/Bank.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A Bank object simulates the behavior of a simple bank/ATM.
 * It contains a Terminal object and two BankAccount objects.
 *
 * Its single public method is open, which 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 1
 */

public class Bank
{
    private String bankName;      // the name of this Bank

    private Terminal atm;         // for talking with the customer

    private BankAccount account1; // two accounts to play with
    private BankAccount account2;

    private static final int INITIAL_BALANCE = 200;
    private static final String HELPSTRING =
        "Transactions: exit, help, deposit, withdraw, balance";

    /**
     * Construct a Bank with the given name.
     * Create two new BankAccounts, each with a starting balance
     * of initialBalance.
     *
     * @param name the name of the Bank.
     */

    public Bank( String name )
    {
        bankName = name;
        atm      = new Terminal();
        account1 = new BankAccount( INITIAL_BALANCE );
        account2 = new BankAccount( INITIAL_BALANCE );
    }

    /**
     * Open the Bank for business.
     *
     * Send a whichAccount message prompting for a BankAccount
     * number, then send a processTransactionsForAccount
     * message to do the work.
     */

```



```
    public void open()
    {
        atm.println( "Welcome to " + bankName );
        boolean bankIsOpen = true;
        while ( bankIsOpen ) {
            BankAccount account = this.whichAccount();
            if ( account == null ) {
                bankIsOpen = false;
            }
            else {
                this.processTransactionsForAccount(account);
            }
        }
        atm.println( "Goodbye from " + bankName );
    }

    // Prompt the user for an account number and return the
    // corresponding BankAccount object. Return null when
    // the Bank is about to close.

    private BankAccount whichAccount()
    {
        int  accountNumber =
            atm.readInt("Account number (1 or 2), 0 to shut down: ");

        if ( accountNumber == 1 ) {
            return account1;
        }
        else if ( accountNumber == 2 ) {
            return account2;
        }
        else if ( accountNumber == 0 ) {
            return null;
        }
        else {
            atm.println( "No account numbered " +
                                accountNumber + "; try again" );
            return this.whichAccount();
        }
    }

    // Prompt the user for transaction to process.
    // Then send an appropriate message to account.

    private void processTransactionsForAccount( BankAccount account)
    {
        atm.println( HELPSTRING );

        boolean moreTransactions = true;
        while ( moreTransactions ) {
            String command = atm.readWord( "transaction: " );
            if ( command.equals( "exit" ) ) {
                moreTransactions = false;
            }
            else if ( command.equals( "help" ) ) {
                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  }
```

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



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

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



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

```
 1  // 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  }
```

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

/**
 * A Box has a width, a height and a paintChar used
 * used to paint the Box on a Screen.
 *
 * Examples:
 * <pre>
 *  new Box( 3, 4, 'G' )   new Box( 1, 1, '$' )
 *
 *          GGG                     $
 *          GGG
 *          GGG
 *          GGG
 * </pre>
 *
 * @version 3
 */

public class Box
{
    private int  width;      // width  in (character) pixels
    private int  height;     // height in (character) pixels
    private char paintChar;  // character used for painting

    /**
     * Construct a box.
     *
     * @param width width in (character) pixels.
     * @param height height in (character) pixels.
     * @param paintChar character used for painting this Box.
     */

    public Box( int width, int height, char paintChar )
    {
        this.width     = width;
        this.height    = height;
        this.paintChar = paintChar;
    }

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

    public void paintOn( Screen s, int x, int y )
    {
        HLine hline = new HLine( width, paintChar );
        for ( int i = 0; i < height; i++ ) {
            hline.paintOn( s, x, y+i );
```



```
        }
    }

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

    public void paintOn( Screen s )
    {
        paintOn( s, 0, 0 );  // or this.paintOn(s,0,0);
    }

    /**
     * Get the width of this Box.
     *
     * @return width of box (expressed as a number
     *         of characters).
     */

    public int getWidth()
    {
        return width;
    }

    /**
     * Get the height of this Box.
     *
     * @return the height in (character) pixels.
     */

    public int getHeight()
    {
        return height;
    }

    /**
     * Set the width of this Box.
     *
     * @param width the new width in (character) pixels.
     */

    public void setWidth( int width )
    {
        this.width = width;
    }

    /**
     * Set the height of this Box.
     *
     * @param height the new height in (character) pixels.
     */

    public void setHeight( int height )
    {
```

```
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 height: ");
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 }
```

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



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

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



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

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



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

```
 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  }
```

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



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

```
// 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 }
```

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

/**
 * A ShoppingCart keeps track of a customer's purchases.
 *
 * @see EStore
 * @version 4
 */

public class ShoppingCart
{
    /// replace these two fields by a single ArrayList
    private int count; // number of Items in this ShoppingCart
    private int cost;  // total cost of Items in this ShoppingCart

    /**
     * Construct a new empty ShoppingCart.
     */

    public ShoppingCart()
    {
        count = 0;
        cost  = 0;
    }

    /**
     * Add an Item to this ShoppingCart.
     *
     * @param item the Item to add.
     */

    public void addItem( Item item )
    {
        /// this code just keeps track of the totals
        /// replace it with code that adds the item to the list
        count++;
        this.cost += item.getCost(); // Java idiom: a += b means a = a +
    }

    /**
     * Return an Item from this ShoppingCart.
     *
     * @param item the Item to return.
     */

    public void returnItem( Item item )
    {
        /// look through the list looking for Item
        /// remove it if it's there
    }

    /**
     * What happens when this ShoppingCart is asked how many
```



```
     * Items it contains.
     *
     * @return the number of items in this ShoppingCart.
     */

    public int getCount()
    {
        /// get this information from the list,
        /// since the count field no longer exists
        return count;
    }

    /**
     * What happens when this ShoppingCart is asked the total
     * cost of the Items it contains.
     *
     * @return the total cost of the items in this ShoppingCart.
     */
    public int getCost()
    {
        /// get this information from the list,
        /// since the cost field no longer exists
        return cost;
    }

    /**
     * Write the contents of this ShoppingCart to a Terminal.
     *
     * @param t the Terminal to use for output.
     */

    public void showContents( Terminal t)
    {
        /// work to do here ...
        t.println("  [sorry, can't yet print ShoppingCart contents]");
    }
}
```

```
 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  }
```

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



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

```
 1  // 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  }
```

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



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

```
 1  // joi/5/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 5
23   */
24
25  public abstract class JFile
26  {
27      /**
28       * The separator used in pathnames.
29       */
30
31      public static final String separator = File.separator;
32
33      private String    name;       // a JFile knows its name
34      private String    owner;      // the owner of this file
35      private Date      createDate; // when this file was created
36      private Date      modDate;    // when this file was last modified
37      private Directory parent;     // the Directory containing this file
38
39      /**
40       * Construct a new JFile, set owner, parent, creation and
41       * modification dates. Add this to parent (unless this is the
42       * root Directory).
43       *
44       * @param name     the name for this file (in its parent directory).
45       * @param creator  the owner of this new file.
46       * @param parent   the Directory in which this file lives.
47       */
48
49      protected JFile( String name, String 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( 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  }
```

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



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

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



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

```
// 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  }
```