

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

1 // fo1/5/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[] getFileNamees()
94     {
95         return (String[])jfiles.keySet().toArray( new String[0] );
96     }
97 }

```



```

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

```

```

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

```

```

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

```

```

57         month = new Month();
58     }
59
60     /**
61     * Simulates interaction with a Bank.
62     * Presents the user with an interactive loop, prompting for
63     * banker transactions and in case of the banker transaction
64     * "customer", an account id and further customer
65     * transactions.
66     */
67
68     public void visit()
69     {
70         instructUser();
71
72         String command;
73         while ( !command =
74             atm.readWord("banker command: ").equals("exit") ) {
75
76             if (command.startsWith("h") ) {
77                 help( BANKER_COMMANDS );
78             }
79             else if (command.startsWith("o") ) {
80                 openNewAccount();
81             }
82             else if (command.startsWith("r") ) {
83                 report();
84             }
85             else if (command.startsWith("c" ) ) {
86                 BankAccount acct = whichAccount();
87                 if ( acct != null )
88                     processTransactionsForAccount( acct );
89             }
90             else {
91                 // Unrecognized Request
92                 atm.println( "unknown command: " + command );
93             }
94         }
95         report();
96         atm.println( "Goodbye from " + bankName );
97     }
98
99     // Open a new bank account,
100     // prompting the user for information.
101     private void openNewAccount()
102     {
103         String accountName = atm.readWord( "Account name: " );
104         char accountType =
105             atm.readChar( "Checking/Fee/Regular? (c/F/r): " );
106         int startup = atm.readInt( "Initial deposit: " );
107         BankAccount newAccount;
108         switch( accountType ) {
109             case 'c':
110                 newAccount = new CheckingAccount( startup, this );
111             case 'f':
112                 newAccount = new FeeAccount( 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     String transaction;
136     while ( !(transaction =
137            atm.readWord(" transaction: ")).equals("quit")) {
138
139         if ( transaction.startsWith( "h" ) ) {
140             help( CUSTOMER_TRANSACTIONS );
141         }
142         else if ( transaction.startsWith( "d" ) ) {
143             int amount = atm.readInt( " amount: " );
144             atm.println( " deposited " + acct.deposit( amount ) );
145         }
146         else if ( transaction.startsWith( "w" ) ) {
147             int amount = atm.readInt( " amount: " );
148             atm.println( " withdrew " + acct.withdraw( amount ) );
149         }
150         else if ( transaction.startsWith( "c" ) ) {
151             int amount = atm.readInt( " amount of check: " );
152             atm.println( " cashed check for " +
153                ((CheckingAccount)acct).honorCheck( amount ) )
154         }
155         else if ( transaction.startsWith( "t" ) ) {
156             atm.print( " to " );
157             BankAccount toacct = whichAccount();
158             if ( toacct != null ) {
159                 int amount = atm.readInt( " amount to transfer: " );
160                 atm.println( " transferred " +
161                    toacct.deposit(acct.withdraw(amount)));
162             }
163         }
164         else if ( transaction.startsWith( "b" ) ) {
165             atm.println( " current balance " +
166                acct.requestBalance());
167         }
168     }

```

```

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

```

```

225
226 // Welcome the user to the bank and instruct her on
227 // her options.
228
229 private void instructUser()
230 {
231     atm.println( "Welcome to " + bankName );
232     atm.println( "Open some accounts and work with them." );
233     help( BANKER_COMMANDS );
234 }
235
236 // Display a help string.
237
238 private void help( String helpString )
239 {
240     atm.println( helpString );
241     atm.println();
242 }
243
244 /**
245  * Increment bank balance by given amount.
246  * @param amount the amount increment.
247  */
248
249 public void incrementBalance(int amount)
250 {
251     balance += amount;
252 }
253
254 /**
255  * Increment by one the count of transactions,
256  * for this bank.
257  */
258
259 public void countTransaction()
260 {
261     transactionCount++;
262 }
263
264 /**
265  * Get the number of transactions performed by this bank.
266  */
267
268 * @return number of transactions performed.
269 */
270
271 public int getTransactionCount()
272 {
273     return transactionCount ;
274 }
275
276 /**
277  * Get the current bank balance.
278  */
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
302 * <p>
303  * A -e argument causes the input to be echoed.
304  * This can be useful for executing the program against
305  * a test script, e.g.,
306  * java Bank -e < Bank.in
307  * </pre>
308
309 * @param args the command line arguments:
310 *     <pre>
311 *     -e echo input.
312 *     bankName any other command line argument.
313 *     </pre>
314 */
315
316 public static void main( String[] args )
317 {
318     // parse the command line arguments for the echo
319     // flag and the name of the bank
320
321     boolean echo = false; // default does not echo
322     String bankName = "Falthless Trust"; // default bank name
323
324     for (int i = 0; i < args.length; i++) {
325         if (args[i].equals("-e")) {
326             echo = true;
327         }
328         else {
329             bankName = args[i];
330         }
331     }
332     Bank aBank = new Bank( bankName, new Terminal(echo) );
333     aBank.visit();
334 }
335

```

```

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

```

```

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

```

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

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

```

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

```



```

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

```

```

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

```

```

57     /**
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 }

```