

```

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
102     private void openNewAccount()
103     {
104         String accountName = atm.readWord( "Account name: " );
105         char accountType =
106             atm.readChar( "Checking/Fee/Regular? (c/F/r): " );
107         int startup = atm.readInt( "Initial deposit: " );
108         BankAccount newAccount;
109         switch( accountType ) {
110             case 'c':
111                 newAccount = new CheckingAccount( startup, this );
112

```

```

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

```

```

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

```

```

225
226 // Welcome the user to the bank and instruct her on
227 // her options.
228
229 private void instructUser()
230 {
231     atm.println( "Welcome to " + bankName );
232     atm.println( "Open some accounts and work with them." );
233     help( BANKER_COMMANDS );
234 }
235
236 // Display a help string.
237
238 private void help( String helpString )
239 {
240     atm.println( helpString );
241     atm.println();
242 }
243
244 /**
245  * Increment bank balance by given amount.
246  * @param amount the amount increment.
247  */
248
249 public void incrementBalance(int amount)
250 {
251     balance += amount;
252 }
253
254 /**
255  * Increment by one the count of transactions,
256  * for this bank.
257  */
258
259 public void countTransaction()
260 {
261     transactionCount++;
262 }
263
264 /**
265  * Get the number of transactions performed by this bank.
266  */
267
268 * @return number of transactions performed.
269 */
270
271 public int getTransactionCount()
272 {
273     return transactionCount ;
274 }
275
276 /**
277  * Get the current bank balance.
278  */
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 // jol/5/bank/class Month
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 import java.io.*;
7 import java.util.Calendar;
8
9 /**
10  * The Month class implements an object that keeps
11  * track of the month of the year.
12  *
13  * @version 5
14  */
15
16 public class Month
17 {
18     private static final String[] monthName =
19         {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
20          "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
21
22     private int month;
23     private int year;
24
25     /**
26      * Month constructor constructs a Month object
27      * initialized to the current month and year.
28      */
29
30     public Month()
31     {
32         Calendar rightNow = Calendar.getInstance();
33         month = rightNow.get( Calendar.MONTH );
34         year = rightNow.get( Calendar.YEAR );
35     }
36
37     /**
38      * Advance to next month.
39      */
40
41     public void next()
42     {
43         // needs completion
44     }
45
46     /**
47      * How a Month is displayed as a String -
48      * for example, "Jan, 2003".
49      */
50     * @return String representation of the month.
51     */
52
53     //
54     //
55     //
56     {
57         public String toString()

```

```

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

```