

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```

```

393     /**
394     * Get the current bank balance.
395     */
396     * @return current bank balance.
397     */
398     public int getBalance( )
399     {
400         return balance;
401     }
402     /**
403     * Get the current number of open accounts.
404     */
405     * @return number of open accounts.
406     */
407     public int getNumberOfAccounts( )
408     {
409         return accountList.size();
410     }
411     /**
412     * Run the simulation by creating and then visiting a new Bank.
413     * <p>
414     * A -e argument causes the input to be echoed.
415     * This can be useful for executing the program against
416     * a test script, e.g.,
417     * <pre>
418     * java Bank -e < Bank.in
419     * </pre>
420     * @param args the command line arguments:
421     *     <pre>
422     *     -e echo input.
423     *     bankName any other command line argument.
424     * </pre>
425     */
426     public static void main( String[] args )
427     {
428         // parse the command line arguments for the echo
429         // flag and the name of the bank
430         boolean echo = false;
431         String bankName = "River Bank"; // default bank name
432         for (int i = 0; i < args.length; i++ ) {
433             if (args[i].equals("-e")) {
434                 echo = true;
435             }
436             else {
437                 bankName = args[i];
438             }
439         }
440     }
441 }
442 }
443 }
444 }
445 }
446 }
447 }
448 }

```

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