

```

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

```

```

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

```

```

169     }
170     else if ( transaction.startsWith( "w" ) ) {
171         int amount = readPosAmt( " amount:" );
172         atm.println( " withdrew "
173             + acct.withdraw( amount ) );
174     }
175     else if ( transaction.startsWith( "c" ) ) {
176         int amount = readPosAmt( " amount of check: " );
177         try { // to cast acct to CheckingAccount ...
178             atm.println( " cashed check for " +
179                 ((CheckingAccount) acct).honorCheck( amount ) )
180         }
181         catch (ClassCastException e) {
182             // if not a checking account, report error
183             atm.errPrintln(
184                 " Sorry, not a checking account. " );
185         }
186     }
187     else if (transaction.startsWith("t")) {
188         atm.print( " to ");
189         BankAccount toacct = whichAccount();
190         if (toacct != null) {
191             int amount = readPosAmt(" amount to transfer: ");
192             atm.println(" 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 exception) {
244             atm.errPrintln(
245                 "Insufficient funds in account \"" +
246                 name + "\" for monthly fee" );
247         }
248     }
249 }
250
251 // Report bank activity. For each BankAccount,
252 // print the customer id (name or number), balance, and
253 // the number of transactions. Then print Bank totals.
254
255 private void report()
256 {
257     atm.println( bankName + " report for " + month );
258     atm.println( "\nSummaries of individual accounts:" );
259     atm.println( "account balance transaction count" );
260     for (Iterator i = accountList.keySet().iterator();
261          i.hasNext(); ) {
262         String accountName = (String) i.next();
263         BankAccount acct = (BankAccount) accountList.get(accountName)
264         atm.println(accountName + "\t$" + acct.getBalance() + "\t\t"
265             + acct.getTransactionCount());
266     }
267     atm.println( "\nBank totals" );
268     atm.println( "open accounts: " + getNumberOfAccounts() );
269     atm.println( "cash on hand: $" + getBalance() );
270     atm.println( "transactions: " + getTransactionCount() );
271     atm.println();
272 }
273
274 // 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 public void incrementBalance(int amount)
313 {
314     balance += amount;
315 }
316
317 /**
318  * Increment by one the count of transactions,
319  * for this bank.
320  */
321 public void countTransaction()
322 {
323     transactionCount++;
324 }
325
326 /**
327  * Get the number of transactions performed by this bank.
328  *
329  * @return number of transactions performed.
330  */
331 public int getTransactionCount()
332 {
333     return transactionCount ;
334 }
335
336

```

```

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 }
```

```

1 // fo1/7/bank/BankAccount.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A BankAccount object has private fields to keep track
8  * of its current balance, the number of transactions
9  * performed and the Bank in which it is an account, and
10 * and public methods to access those fields appropriately.
11 *
12 * @see Bank
13 * @version 7
14 */
15
16 public abstract class BankAccount
17 {
18     private int balance = 0; // Account balance (whole dollars)
19     private int transactionCount = 0; // Number of transactions 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      * @exception InsufficientFundsException when appropriate.
31      */
32     protected BankAccount( int initialBalance, Bank issuingBank )
33     throws InsufficientFundsException
34     {
35         this.issuingBank = issuingBank;
36         deposit( initialBalance );
37     }
38
39     /**
40      * Get transaction fee. By default, 0.
41      *
42      * Override this for accounts having transaction fees.
43      *
44      * @return the fee.
45      */
46     protected int getTransactionFee()
47     {
48         return 0;
49     }
50
51     /**
52      * The bank that issued this account.
53      *
54      * @return the Bank.
55      */
56

```

```

57     protected Bank getIssuingBank()
58     {
59         return issuingBank;
60     }
61
62     /**
63      * Withdraw the given amount, decreasing this BankAccount's
64      * balance and the issuing Bank's balance.
65      *
66      * Counts as a transaction.
67      *
68      * @param amount the amount to be withdrawn
69      * @return amount withdrawn
70      *
71      * @exception InsufficientFundsException when appropriate.
72      */
73
74     public int withdraw( int amount )
75     throws InsufficientFundsException
76     {
77         incrementBalance( -amount - getTransactionFee() );
78         countTransaction();
79         return amount ;
80     }
81
82     /**
83      * Deposit the given amount, increasing this BankAccount's
84      * balance and the issuing Bank's balance.
85      *
86      * Counts as a transaction.
87      *
88      * @param amount the amount to be deposited
89      * @return amount deposited
90      *
91      * @exception InsufficientFundsException when appropriate.
92      */
93     public int deposit( int amount )
94     throws InsufficientFundsException
95     {
96         incrementBalance( amount - getTransactionFee() );
97         countTransaction();
98         return amount ;
99     }
100
101     /**
102      * Request for balance. Counts as a transaction.
103      *
104      * @return current account balance.
105      *
106      * @exception InsufficientFundsException when appropriate.
107      */
108
109     public int requestBalance()
110     throws InsufficientFundsException
111     {
112         incrementBalance( - getTransactionFee() );

```

```

113     countTransaction();
114     return getBalance() ;
115 }
116
117 /**
118  * Get the current balance.
119  * Does NOT count as a transaction.
120  */
121     @return current account balance
122     */
123     public int getBalance()
124     {
125         return balance;
126     }
127
128 /**
129  * Increment account balance by given amount.
130  * Also increment issuing Bank's balance.
131  * Does NOT count as a transaction.
132  */
133     * @param amount the amount of the increment.
134     * @exception InsufficientFundsException when appropriate.
135     */
136     public final void incrementBalance( int amount )
137     {
138         throws InsufficientFundsException
139         {
140             int newBalance = balance + amount;
141             if (newBalance < 0) {
142                 throw new InsufficientFundsException(
143                     "For this transaction" );
144             }
145             balance = newBalance;
146             getIssuingBank().incrementBalance( amount );
147         }
148     }
149
150 /**
151  * Get the number of transactions performed by this
152  * account. Does NOT count as a transaction.
153  */
154     * @return number of transactions performed.
155     */
156     public int getTransactionCount()
157     {
158         return transactionCount;
159     }
160 }
161
162 /**
163  * Increment by 1 the count of transactions, for this account
164  * and for the issuing Bank.
165  * Does NOT count as a transaction.
166  */
167     * @exception InsufficientFundsException when appropriate.
168

```

```

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

```

```

1 // fo1/7/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 * @see BankAccount
12 *
13 * @version 7
14 */
15
16 public class CheckingAccount extends BankAccount
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      * @exception InsufficientFundsException when appropriate.
27      */
28
29     public CheckingAccount( int initialBalance, Bank issuingBank )
30     throws InsufficientFundsException
31     {
32         super( initialBalance, issuingBank );
33     }
34
35     /**
36      * Honor a check:
37      * Charge the account the appropriate fee
38      * and withdraw the amount.
39      *
40      * @param amount amount (in whole dollars) to be withdrawn.
41      * @return the amount withdrawn.
42      *
43      * @exception InsufficientFundsException when appropriate.
44      */
45
46     public int honorCheck( int amount )
47     throws InsufficientFundsException
48     {
49         // careful error checking logic:
50         // first try to deduct the check fee
51         // if you succeed, try to honor check
52         // if that fails, remember to add back the check fee!
53
54         try {
55             incrementBalance( - getIssuingBank().getCheckFee() );
56

```

```

57         catch (InsufficientFundsException e) {
58             throw new InsufficientFundsException(
59                 "to cover check fee" );
60         }
61         try {
62             withdraw( amount );
63         }
64         catch (InsufficientFundsException e) {
65             incrementBalance( getIssuingBank().getCheckFee() );
66             throw new InsufficientFundsException(
67                 "to cover check + check fee" );
68         }
69         return amount;
70     }
71
72     /**
73      * Nothing special happens to a CheckingAccount on the
74      * first day of the month.
75      */
76     public void newMonth()
77     {
78         return;
79     }
80
81 }

```



```

1 // fo1/7/bank/SavingsAccount.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * A SavingsAccount is a BankAccount that bears interest.
8  * A fee is charged for too many transactions in a month.
9  *
10 * @see BankAccount
11 *
12 * @version 7
13 */
14
15 public class SavingsAccount extends BankAccount
16 {
17     private int transactionsThisMonth;
18
19     /**
20      * Override getTransactionFee() to return a non-zero fee
21      * after the appropriate number of free monthly transactions.
22      *
23      * @return the fee for current transaction.
24      */
25     protected int getTransactionFee()
26     {
27         if (transactionsThisMonth >
28             getIssuingBank().getMaxFreeTransactions()) {
29             return getIssuingBank().getTransactionFee();
30         }
31         else {
32             return 0;
33         }
34     }
35
36     /**
37      * Increment count of transactions, for this account for
38      * this Month and in total and for the issuing Bank, by one.
39      *
40      * @exception InsufficientFundsException when appropriate.
41      */
42     public void countTransaction()
43     throws InsufficientFundsException
44     {
45         transactionsThisMonth++;
46         super.countTransaction();
47     }
48
49     /**
50      * Constructor, accepting an initial balance.
51      * @param initialBalance the opening balance.
52      * @param issuingBank the bank that issued this account.
53      */
54
55
56

```

```

57     * @exception InsufficientFundsException when appropriate.
58     */
59
60     public SavingsAccount( int initialBalance, Bank issuingBank )
61     throws InsufficientFundsException
62     {
63         super( initialBalance, issuingBank);
64         transactionsThisMonth = 1;
65     }
66
67     /**
68      * A SavingsAccount earns interest each month.
69      *
70      * @exception InsufficientFundsException when appropriate.
71      */
72     public void newMonth()
73     throws InsufficientFundsException
74     {
75         double monthlyRate = getIssuingBank().getInterestRate()/12;
76         incrementBalance( (int)(monthlyRate * getBalance()));
77         transactionsThisMonth = 0;
78     }
79
80 }

```

```

1 // fo1/7/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 * @see BankAccount
11 *
12 * @version 7
13 */
14
15 public class FeeAccount extends BankAccount
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      * @exception InsufficientFundsException when appropriate.
24      */
25
26     public FeeAccount( int initialBalance, Bank issuingBank )
27     throws InsufficientFundsException
28     {
29         super( initialBalance, issuingBank);
30     }
31
32     /**
33      * The Bank's transaction fee.
34      *
35      * @return the fee.
36      */
37
38     protected int getTransactionFee()
39     {
40         return getIssuingBank().getTransactionFee();
41     }
42
43     /**
44      * The way a transaction is counted for a FeeAccount: it levies
45      * a transaction fee as well as counting the transaction.
46      *
47      * @exception InsufficientFundsException when appropriate.
48      */
49
50     public void countTransaction()
51     throws InsufficientFundsException
52     {
53         incrementBalance( - getTransactionFee() );
54         super.countTransaction();
55     }
56

```

```

57     /**
58      * A FeeAccount incurs a monthly charge.
59      *
60      * @exception InsufficientFundsException when appropriate.
61      */
62
63     public void newMonth()
64     throws InsufficientFundsException
65     {
66         incrementBalance( - getIssuingBank().getMonthlyCharge());
67     }
68 }

```

```
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     * @exception InsufficientFundsException when appropriate.
24     */
25
26     public RegularAccount( int initialBalance, Bank issuingBank )
27     throws InsufficientFundsException
28     {
29         super( initialBalance, issuingBank );
30     }
31
32     /**
33     * Action to take when a new month starts.
34     *
35     * A RegularAccount does nothing when the next month starts.
36     */
37
38     public void newMonth() {
39         // do nothing
40     }
41
42 }
```

```

1 // foj/7/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 7
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         month = (month + 1) % 12;
44         if (month == 0) {
45             year++;
46         }
47     }
48
49     /**
50      * How a Month is displayed as a String -
51      * for example, "Jan, 2003".
52      *
53      * @return String representation of the month.
54      */
55     public String toString()

```

```

57     {
58         return monthName[month] + ", " + year;
59     }
60
61     /**
62      * For unit testing.
63      */
64
65     public static void main( String[] args )
66     {
67         Month m = new Month();
68         for (int i=0; i < 14; i++, m.next()) {
69             System.out.println(m);
70         }
71         for (int i=0; i < 35; i++, m.next()) { // no loop body
72             System.out.println( "three years later: " + m );
73             for (int i=0; i < 120; i++, m.next()) { // no loop body
74                 System.out.println( "ten years later: " + m );
75             }
76         }

```

```
1 // fo1/7/bank/InsufficientFundsException.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * Thrown when there is an attempt to spend money that is not there.
8  *
9  * @version 7
10 */
11
12 public class InsufficientFundsException extends Exception
13 {
14     /**
15      * Construct an InsufficientFundsException
16      * with a String description.
17      *
18      * @param msg a more specific description.
19      */
20
21     public InsufficientFundsException( String msg )
22     {
23         super( msg );
24     }
25
26     /**
27      * Construct an InsufficientFundsException
28      * with no description.
29      */
30
31     public InsufficientFundsException()
32     {
33         this( "" );
34     }
35 }
```

```
1 // foj/7/bank/NegativeAmountException.java
2 //
3 //
4 // Copyright 2003 Bill Campbell and Ethan Bolker
5
6 /**
7  * Thrown when attempting to work with a negative amount.
8  *
9  * @version 7
10 */
11
12 public class NegativeAmountException extends Exception
13 {
14 }
```