```
// joi/7/bank/Bank.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * A Bank object simulates the behavior of a simple bank/ATM.
 * It contains a Terminal object and a collection of
 * BankAccount objects.
 *
 * The visit method opens this Bank for business,
 * prompting the customer for input.
 *
 * To create a Bank and open it for business issue the command
 * <code>java Bank</code>.
 *
 * @see BankAccount
 * @version 7
 */

public class Bank
{
    private String bankName;           // the name of this Bank
    private Terminal atm;              // for talking with the customer
    private int balance = 0;           // total cash on hand
    private int transactionCount = 0;  // number of Bank transactions
    private Month month;               // the current month.
    private Map accountList;           // mapping names to accounts.

    private int checkFee = 2;            // cost for each check
    private int transactionFee = 1;      // fee for each transaction
    private int monthlyCharge = 5;       // monthly charge
    private double interestRate = 0.05;  // annual rate paid on savings
    private int maxFreeTransactions = 3; // for savings accounts

    // what the banker can ask of the bank

    private static final String BANKER_COMMANDS =
    "Banker commands: " +
        "exit, open, customer, nextmonth, report, help.";

    // what the customer can ask of the bank

    private static final String CUSTOMER_TRANSACTIONS =
    "    Customer transactions: " +
    "deposit, withdraw, transfer, balance, cash check, quit, help.";

    /**
     * Construct a Bank with the given name and Terminal.
     *
     * @param bankName the name for this Bank.
     * @param atm  this Bank's Terminal.
     */

```



```
    public Bank( String bankName, Terminal atm )
    {
        this.atm      = atm;
        this.bankName = bankName;
        accountList   = new TreeMap();
        month         = new Month();
    }

    /**
     * Simulates interaction with a Bank.
     * Presents the user with an interactive loop, prompting for
     * banker transactions and in the case of the banker
     * transaction "customer", an account id and further
     * customer transactions.
     */

    public void visit()
    {
        instructUser();

        String command;
        while (!(command =
                 atm.readWord("banker command: ")).equals("exit")) {

            if (command.startsWith("h")) {
                help( BANKER_COMMANDS );
            }
            else if (command.startsWith("o")) {
                openNewAccount();
            }
            else if (command.startsWith("n")) {
                newMonth();
            }
            else if (command.startsWith("r")) {
                report();
            }
            else if (command.startsWith( "c" ) ) {
                BankAccount acct = whichAccount();
                if ( acct != null ) {
                        processTransactionsForAccount( acct );
                }
            }
            else {
                // Unrecognized Request
                atm.println( "unknown command: " + command );
            }
        }
        report();
        atm.println( "Goodbye from " + bankName );
    }


    // Open a new bank account,
    // prompting the user for information.

    private void openNewAccount()
```

```
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("    transfered "
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
215      // Prompt for an account name (or number), look it up
216      // in the account list. If it's there, return it;
217      // otherwise report an error and return null.
218
219      private BankAccount whichAccount()
220      {
221          String accountName = atm.readWord( "account name: " );
222          BankAccount account = (BankAccount) accountList.get(accountName);
223          if (account == null) {
224              atm.println( "not a valid account" );
```

```
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
275      // Welcome the user to the bank and instruct her on
276      //  her options.
277
278      private void instructUser()
279      {
280          atm.println( "Welcome to " + bankName );
```



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



```
393
394      /**
395       * Get the current bank balance.
396       *
397       * @return current bank balance.
398       */
399
400      public int getBalance()
401      {
402          return balance;
403      }
404
405      /**
406       * Get the current number of open accounts.
407       *
408       * @return number of open accounts.
409       */
410
411      public int getNumberOfAccounts()
412      {
413          return accountList.size();
414      }
415
416      /**
417       * Run the simulation by creating and then visiting a new Bank.
418       * <p>
419       * A -e argument causes the input to be echoed.
420       * This can be useful for executing the program against
421       * a test script, e.g.,
422       * <pre>
423       *   java Bank -e < Bank.in
424       * </pre>
425       *
426       * @param args the command line arguments:
427       *         <pre>
428       *         -e echo input.
429       *         bankName any other command line argument.
430       *         </pre>
431       */
432
433      public static void main( String[] args )
434      {
435          // parse the command line arguments for the echo
436          // flag and the name of the bank
437
438          boolean echo    = false;         // default does not echo
439          String bankName = "River Bank";  // default bank name
440
441          for (int i = 0; i < args.length; i++ ) {
442              if (args[i].equals("-e")) {
443                  echo = true;
444              }
445              else {
446                  bankName = args[i];
447              }
448          }
```

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

```
// joi/7/bank/BankAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A BankAccount object has private fields to keep track
 * of its current balance, the number of transactions
 * performed and the Bank in which it is an account, and
 * and public methods to access those fields appropriately.
 *
 * @see Bank
 * @version 7
 */

public abstract class BankAccount
{
    private int  balance = 0;          // Account balance (whole dollars)
    private int  transactionCount = 0; // Number of transactions performe
    private Bank issuingBank;          // Bank issuing this account

    /**
     * Construct a BankAccount with the given initial balance and
     * issuing Bank. Construction counts as this BankAccount's
     * first transaction.
     *
     * @param initialBalance the opening balance.
     * @param issuingBank the bank that issued this account.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    protected BankAccount( int initialBalance, Bank issuingBank )
        throws InsufficientFundsException
    {
        this.issuingBank = issuingBank;
        deposit( initialBalance );
    }

    /**
     * Get transaction fee.  By default, 0.
     * Override this for accounts having transaction fees.
     *
     * @return the fee.
     */

    protected int getTransactionFee()
    {
        return 0;
    }

    /**
     * The bank that issued this account.
     *
     * @return the Bank.
     */
```



```

    protected Bank getIssuingBank()
    {
        return issuingBank;
    }

    /**
     * Withdraw the given amount, decreasing this BankAccount's
     * balance and the issuing Bank's balance.
     * Counts as a transaction.
     *
     * @param amount the amount to be withdrawn
     * @return amount withdrawn
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public int withdraw( int amount )
        throws InsufficientFundsException
    {
        incrementBalance( -amount - getTransactionFee() );
        countTransaction();
        return amount ;
    }

    /**
     * Deposit the given amount, increasing this BankAccount's
     * balance and the issuing Bank's balance.
     * Counts as a transaction.
     *
     * @param amount the amount to be deposited
     * @return amount deposited
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public int deposit(int amount)
        throws InsufficientFundsException
    {
        incrementBalance( amount - getTransactionFee() );
        countTransaction();
        return amount ;
    }

    /**
     * Request for balance. Counts as a transaction.
     *
     * @return current account balance.
     *
     * @exception InsufficientFundsException when appropriate.
     */

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



```
169       */
170
171      public void countTransaction()
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
185      public abstract void newMonth()
186          throws InsufficientFundsException;
187  }
```

```
// joi/7/bank/CheckingAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A CheckingAccount is a BankAccount with one new feature:
 * the ability to cash a check by calling the honorCheck method.
 * Each honored check costs the customer a checkFee.
 *
 * @see BankAccount
 *
 * @version 7
 */

public class CheckingAccount extends BankAccount
{
    /**
     * Constructs a CheckingAccount with the given
     * initial balance and issuing Bank.
     * Counts as this account's first transaction.
     *
     * @param initialBalance the opening balance for this account.
     * @param issuingBank the bank that issued this account.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public CheckingAccount( int initialBalance, Bank issuingBank )
        throws InsufficientFundsException
    {
        super( initialBalance, issuingBank );
    }

    /**
     * Honor a check:
     * Charge the account the appropriate fee
     * and withdraw the amount.
     *
     * @param amount amount (in whole dollars) to be withdrawn.
     * @return the amount withdrawn.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public int honorCheck( int amount )
        throws InsufficientFundsException
    {
        // careful error checking logic:
        // first try to deduct the check fee
        // if you succeed, try to honor check
        // if that fails, remember to add back the check fee!

        try {
            incrementBalance( - getIssuingBank().getCheckFee() );
        }
```



```
        catch (InsufficientFundsException e) {
            throw new InsufficientFundsException(
                        "to cover check fee" );
        }
        try {
            withdraw( amount );
        }
        catch (InsufficientFundsException e) {
            incrementBalance( getIssuingBank().getCheckFee() );
            throw new InsufficientFundsException(
                        "to cover check + check fee" );
        }
        return amount;
    }

    /**
     *  Nothing special happens to a CheckingAccount on the
     *  first day of the month.
     */

    public void newMonth()
    {
        return;
    }
}
```

```
// joi/7/bank/SavingsAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A SavingsAccount is a BankAccount that bears interest.
 * A fee is charged for too many transactions in a month.
 *
 * @see BankAccount
 *
 * @version 7
 */

public class SavingsAccount extends BankAccount
{
    private int transactionsThisMonth;

    /**
     * Override getTransactionFee() to return a non-zero fee
     * after the appropriate number of free monthly transactions.
     *
     * @return the fee for current transaction.
     */

    protected int getTransactionFee()
    {
        if (transactionsThisMonth >
            getIssuingBank().getMaxFreeTransactions()) {
            return getIssuingBank().getTransactionFee();
        }
        else {
            return 0;
        }
    }

    /**
     * Increment count of transactions, for this account for
     * this Month and in total  and for the issuing Bank, by one.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public void countTransaction()
        throws InsufficientFundsException
    {
        transactionsThisMonth++;
        super.countTransaction();
    }

    /**
     * Constructor, accepting an initial balance.
     * @param initialBalance the opening balance.
     *
     * @param issuingBank the bank that issued this account.
     *
```



```
     * @exception InsufficientFundsException when appropriate.
     */

    public SavingsAccount( int initialBalance, Bank issuingBank )
         throws InsufficientFundsException
    {
        super( initialBalance, issuingBank);
        transactionsThisMonth = 1;
    }

    /**
     * A SavingsAccount earns interest each month.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public void newMonth()
         throws InsufficientFundsException
    {
        double monthlyRate = getIssuingBank().getInterestRate()/12;
        incrementBalance( (int)(monthlyRate * getBalance()));
        transactionsThisMonth = 0;
    }
}
```

```
// joi/7/bank/FeeAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A FeeAccount is a BankAccount with one new feature:
 * the user is charged for each transaction.
 *
 * @see BankAccount
 *
 * @version 7
 */

public class FeeAccount extends BankAccount
{
    /**
     * Constructor, accepting an initial balance and issuing Bank.
     *
     * @param initialBalance the opening balance.
     * @param issuingBank the bank that issued this account.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public FeeAccount( int initialBalance, Bank issuingBank )
        throws InsufficientFundsException
    {
        super( initialBalance, issuingBank);
    }

    /**
     * The Bank's transaction fee.
     *
     * @return the fee.
     */

    protected int getTransactionFee()
    {
        return getIssuingBank().getTransactionFee();
    }

    /**
     * The way a transaction is counted for a FeeAccount: it levies
     * a transaction fee as well as counting the transaction.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public void countTransaction()
        throws InsufficientFundsException
    {
        incrementBalance( - getTransactionFee() );
        super.countTransaction();
    }

```



```
    /**
     * A FeeAccount incurs a monthly charge.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public void newMonth()
         throws InsufficientFundsException
    {
        incrementBalance( - getIssuingBank().getMonthlyCharge());
    }
}
```

```
 1  // joi/5/bank/RegularAccount.java
 2  //
 3  //
 4  // Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  /**
 7   * A RegularAccount is a BankAccount that has no special behavior.
 8   *
 9   * It does what a BankAccount does.
10   */
11
12  public class RegularAccount extends BankAccount
13  {
14
15      /**
16       * Construct a BankAccount with the given initial balance and
17       * issuing Bank. Construction counts as this BankAccount's
18       * first transaction.
19       *
20       * @param initialBalance the opening balance.
21       * @param issuingBank the bank that issued this account.
22       *
23       * @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  // joi/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
56      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  // joi/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  // joi/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  }
```