```
 1  // joi/9/copy/Copy1.java
 2  //
 3  //
 4  // Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  import java.io.*;
 7
 8  /**
 9   * Simple read-a-char, write-a-char loop to exercise file I/O.
10   *
11   * Usage: java Copy1 inputfile outputfile
12   */
13
14  public class Copy1
15  {
16      private static final int EOF = -1;  // end of file character rep.
17
18      /**
19       * All work is done here.
20       *
21       * @param args names of the input file and output file.
22       */
23
24      public static void main( String[] args )
25      {
26          FileReader inStream  = null;
27          FileWriter outStream = null;
28          int ch;
29
30          try {
31              // open the files
32              inStream  = new FileReader( args[0] );
33              outStream = new FileWriter( args[1] );
34
35              // copy
36              while ((ch = inStream.read()) != EOF) {
37                  outStream.write( ch );
38              }
39          }
40          catch (IndexOutOfBoundsException e) {
41              System.err.println(
42                  "usage: java Copy1 sourcefile targetfile" );
43          }
44          catch (FileNotFoundException e) {
45              System.err.println( e );  // rely on e's toString()
46          }
47          catch (IOException e) {
48              System.err.println( e );
49          }
50          finally { // close the files
51              try {
52                  if (inStream != null) {
53                      inStream.close();
54                  }
55              }
56              catch (Exception e) {
```



```
57                  System.err.println("Unable to close input stream.");
58              }
59              try {
60                  if (outStream != null) {
61                      outStream.close();
62                  }
63              }
64              catch (Exception e) {
65                  System.err.println("Unable to close output stream.");
66              }
67          }
68      }
69  }
```

```
 1  // joi/9/copy/Copy2.java
 2  //
 3  //
 4  // Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  import java.io.*;
 7
 8  /**
 9   * Simple read-a-line write-a-line loop to exercise file I/O.
10   *
11   * Usage: java Copy2 inputfile outputfile
12   */
13
14  public class Copy2
15  {
16      /**
17       * All work is done here.
18       *
19       * @param args names of the input file and output file.
20       */
21
22      public static void main( String[] args )
23      {
24          BufferedReader inStream  = null;
25          BufferedWriter outStream = null;
26          String line;
27
28          try {
29              // open the files
30              inStream =  new BufferedReader(new FileReader(args[0]));
31              outStream = new BufferedWriter(new FileWriter(args[1]));
32
33              // copy
34              while ((line = inStream.readLine()) != null) {
35                  outStream.write( line );
36                  outStream.newLine();
37              }
38          }
39          catch (IndexOutOfBoundsException e) {
40              System.err.println(
41                  "usage: java Copy2 sourcefile targetfile" );
42          }
43          catch (FileNotFoundException e) {
44              System.err.println( e );  // rely on e's toString()
45          }
46          catch (IOException e) {
47              System.err.println( e );
48          }
49          finally { // close the files
50              try {
51                  if (inStream != null) {
52                      inStream.close();
53                  }
54              }
55              catch (Exception e) {
56                  System.err.println("Unable to close input stream.");
```



```
57              }
58              try {
59                  if (outStream != null) {
60                      outStream.close();
61                  }
62              }
63              catch (Exception e) {
64                  System.err.println("Unable to close output stream.");
65              }
66          }
67      }
68  }
```

```
// joi/9/bank/Bank.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;
import java.io.*;

/**
 * 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.
 *
 * It is persistent: it can save its state to a file and read it
 * back at a later time.
 *
 * To create a Bank and open it for business issue the command
 * <code>java Bank</code> with appropriate arguments.
 *
 * @see BankAccount
 * @version 9
 */

public class Bank
    implements Serializable
{
    private String bankName;           // the name of this Bank
    private transient Terminal atm;   // for communication with world
    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,\n" +
    "     balance, cash check, quit, help.";

    /**
     * Construct a Bank with the given name.
```



```
     *
     * @param bankName the name for this Bank.
     */

    public Bank( String bankName )
    {
        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 );
    }
```

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



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

```
225          String accountName = atm.readWord( "account name: " );
226          BankAccount account = (BankAccount) accountList.get(accountName);
227          if (account == null) {
228              atm.println( "not a valid account" );
229          }
230          return account;
231      }
232
233      // Action to take when a new month starts.
234      // Update the month field by sending a next message.
235      // Loop on all accounts, sending each a newMonth message.
236
237      private void newMonth()
238      {
239          month.next();
240          Iterator i = accountList.keySet().iterator();
241          while ( i.hasNext() ) {
242              String name = (String) i.next();
243              BankAccount acct = (BankAccount) accountList.get( name );
244              try {
245                  acct.newMonth();
246              }
247              catch (InsufficientFundsException exception) {
248                  atm.errPrintln( "Insufficient funds in account \"" +
249                                  name + "\" for monthly fee" );
250              }
251          }
252      }
253
254      // Report bank activity.
255      // For each BankAccount, print the customer id (name or number),
256      // account balance and the number of transactions.
257      // Then print Bank totals.
258
259      private void report()
260      {
261          atm.println( bankName + " report for " + month );
262          atm.println( "\nSummaries of individual accounts:" );
263          atm.println( "account  balance   transaction count" );
264          for (Iterator i = accountList.keySet().iterator();
265               i.hasNext(); ) {
266              String accountName = (String) i.next();
267              BankAccount acct = (BankAccount) accountList.get(accountName)
268              atm.println(accountName + "\t$" + acct.getBalance() + "\t\t"
269                          + acct.getTransactionCount());
270          }
271          atm.println( "\nBank totals");
272          atm.println( "open accounts: " + getNumberOfAccounts() );
273          atm.println( "cash on hand: $" + getBalance());
274          atm.println( "transactions:  " + getTransactionCount());
275          atm.println();
276      }
277
278
279      // Welcome the user to the bank and instruct her on
280      //  her options.
```



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

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



```
393      public int getMaxFreeTransactions()
394      {
395          return maxFreeTransactions;
396      }
397
398      /**
399       * Get the current bank balance.
400       *
401       * @return current bank balance.
402       */
403
404      public int getBalance()
405      {
406          return balance;
407      }
408
409      /**
410       * Get the current number of open accounts.
411       *
412       * @return number of open accounts.
413       */
414
415      public int getNumberOfAccounts()
416      {
417          return accountList.size();
418      }
419
420      /**
421       * Set the atm for this Bank.
422       *
423       * @param atm the Bank's atm.
424       */
425
426      public void setAtm( Terminal atm ) {
427          this.atm = atm;
428      }
429
430
431      /**
432       * Run the simulation by creating and then visiting a new Bank.
433       * <p>
434       * A -e argument causes the input to be echoed.
435       * This can be useful for executing the program against
436       * a test script, e.g.,
437       * <pre>
438       *   java Bank -e < Bank.in
439       * </pre>
440       *
441       * The -f argument reads the bank's state from the specified
442       * file, and writes it to that file when the program exits.
443       *
444       * @param args the command line arguments:
445       *         <pre>
446       *         -e echo input.
447       *         -f filename
448       *         bankName any other command line argument.
```

```
449       *         </pre>
450       */
451
452      public static void main( String[] args )
453      {
454          boolean echo        = false;
455          String bankFileName = null;
456          String bankName     = "Persistent Bank";
457          Bank theBank        = null;
458
459          // parse the command line arguments
460          for (int i = 0; i < args.length; i++ ) {
461              if (args[i].equals("-e")) { // echo input to output
462                  echo = true;
463                  continue;
464              }
465              if (args[i].equals("-f")) { // read/write Bank from/to file
466                  bankFileName = args[++i];
467                  continue;
468              }
469          }
470
471          // create a new Bank or read one from a file
472          if (bankFileName == null) {
473              theBank = new Bank( bankName );
474          }
475          else {
476              theBank = readBank( bankName, bankFileName );
477          }
478
479          // give the Bank a Terminal, then visit
480          theBank.setAtm(new Terminal(echo));
481          theBank.visit();
482
483          // write theBank's state to a file if required
484          if (bankFileName != null) {
485              writeBank(theBank, bankFileName);
486          }
487      }
488
489      // Read a Bank from a file (create it if file doesn't exist).
490      //
491      // @param bankName     the name of the Bank
492      // @param bankFileName the name of the file containing the Bank
493      //
494      // @return the Bank
495
496      private static Bank readBank(String bankName, String bankFileName)
497      {
498          File file = new File( bankFileName );
499          if (!file.exists()) {
500              return new Bank( bankName );
501          }
502          ObjectInputStream inStream = null;
503          try {
504              inStream = new ObjectInputStream(
```



```
505                              new FileInputStream( file ) );
506              Bank bank = (Bank)inStream.readObject();
507              System.out.println(
508                 "Bank state read from file " + bankFileName);
509              return bank;
510          }
511          catch (Exception e ) {
512              System.err.println(
513                  "Problem reading " + bankFileName );
514              System.err.println(e);
515              System.exit(1);
516          }
517          finally {
518              try {
519                  inStream.close();
520              }
521              catch (Exception e) {
522              }
523          }
524          return null; // you can never get here
525      }
526
527
528      // Write a Bank to a file.
529      //
530      // @param bank     the Bank
531      // @param fileName the name of the file to write the Bank to
532
533      private static void writeBank( Bank bank, String fileName)
534      {
535          ObjectOutputStream outStream = null;
536          try {
537              outStream = new ObjectOutputStream(
538                              new FileOutputStream( fileName ) );
539              outStream.writeObject( bank );
540              System.out.println(
541                 "Bank state written to file " + fileName);
542          }
543          catch (Exception e ) {
544              System.err.println(
545                 "Problem writing " + fileName );
546          }
547          finally {
548              try {
549                  outStream.close();
550              }
551              catch (Exception e ) {
552              }
553          }
554      }
555  }
```

```
// joi/9/bank/BankAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.io.Serializable;

/**
 * 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 9
 */

public abstract class BankAccount
    implements Serializable
{
    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()
```

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



```
169       * Does NOT count as a transaction.
170       *
171       * @exception InsufficientFundsException when appropriate.
172       */
173
174      public void countTransaction()
175          throws InsufficientFundsException
176      {
177          transactionCount++;
178          this.getIssuingBank().countTransaction();
179      }
180
181      /**
182       * Action to take when a new month starts.
183       *
184       * @exception InsufficientFundsException thrown when funds
185       *             on hand are not enough to cover the fees.
186       */
187
188      public abstract void newMonth()
189          throws InsufficientFundsException;
190  }
```

```
// joi/9/bank/class Month
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.io.*;
import java.util.Calendar;

/**
 * The Month class implements an object that keeps
 * track of the month of the year.
 *
 * @version 9
 */

public class Month
    implements Serializable
{
    private static final String[] monthName =
        {"Jan", "Feb", "Mar", "Apr", "May", "Jun",
         "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};

    private int month;
    private int year;

    /**
     *  Month constructor constructs a Month object
     *  initialized to the current month and year.
     */

    public Month()
    {
        Calendar rightNow = Calendar.getInstance();
        month = rightNow.get( Calendar.MONTH );
        year  = rightNow.get( Calendar.YEAR );
    }

    /**
     *  Advance to next month.
     */

    public void next()
    {
        month = (month + 1) % 12;
        if (month == 0) {
            year++;
        }
    }

    /**
     * How a Month is displayed as a String -
     * for example, "Jan, 2003".
     *
     * @return String representation of the month.
     */

```



```
    public String toString()
    {
        return monthName[month] + ", " + year;
    }

    /**
     * For unit testing.
     */

    public static void main( String[] args )
    {
        Month m = new Month();
        for (int i=0; i < 14; i++, m.next()) {
            System.out.println(m);
        }
        for (int i=0; i < 35; i++, m.next()); // no loop body
        System.out.println( "three years later: " + m );
        for (int i=0; i < 120; i++, m.next());// no loop body
        System.out.println( "ten years later: " + m );
    }
}
```