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

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

```java
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             }
```

```java
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    {
```

```java
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     private void newMonth()
237     {
238         month.next();
239
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 transaction id (name or number),
256     // account balance and the number of transactions.
257     // Then print Bank totals.
258     private void report()
259     {
260         atm.println( bankName + " report for " + month );
261         atm.println( "\nSummaries of individual accounts:" );
262         atm.println( "account balance    transaction count" );
263         for (Iterator i = accountList.keySet().iterator();
264             i.hasNext();) {
265             String accountName = (String) i.next();
266             BankAccount acct = (BankAccount) accountList.get(accountName);
267             atm.println(accountName + "\t$" + acct.getBalance() + "\t\t"
268                 + acct.getTransactionCount());
269         }
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     // Welcome the user to the bank and instruct her on
279     // her options.
280
```

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

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

```java
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     * Run the simulation by creating and then visiting a new Bank.
432     * <p>
433     * A -e argument causes the input to be echoed.
434     * This can be useful for executing the program against
435     * a test script, e.g.,
436     * <pre>
437     *   java Bank -e < Bank.in
438     * </pre>
439     *
440     * The -f argument reads the bank's state from the specified
441     * file, and writes it to that file when the program exits.
442     *
443     * @param args the command line arguments:
444     * <pre>
445     *   -e          echo input.
446     *   -f filename
447     *   bankName    any other command line argument.
448     *
```

```
449         */
450          *
451      </pre>
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  // @param bankName      the name of the Bank
491  // @param bankFileName  the name of the file containing the Bank
492  //
493  // @return the Bank
494
495  private static Bank readBank(String bankName, String bankFileName)
496  {
497      File file = new File( bankFileName ) ;
498      if (!file.exists()) {
499          return new Bank( bankName ) ;
500      }
501      ObjectInputStream inStream = null;
502      try {
503          inStream = new ObjectInputStream(
504
```

```
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  // Write a Bank to a file.
528  // @param bank      the Bank
529  // @param fileName  the name of the file to write the Bank to
530  //
531  private static void writeBank( Bank bank, String fileName)
532  {
533      ObjectOutputStream outStream = null;
534      try {
535          outStream = new ObjectOutputStream(
536              new FileOutputStream( fileName ) ) ;
537          outStream.writeObject( bank ) ;
538          System.out.println(
539              "Bank state written to file " + fileName);
540      }
541      catch (Exception e ) {
542          System.err.println(
543              "Problem writing " + fileName ) ;
544      }
545      finally {
546          try {
547              outStream.close();
548          }
549          catch (Exception e ) {
550          }
551      }
552  }
553  }
554  }
555
```