```
// joi/4/bank/Bank.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

// Lines marked "///" flag places where changes will be needed.

/// 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.
 *
 * Its public method visit opens the 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 4
 */

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 don

    private BankAccount[] accountList; /// collection of BankAccounts
    /// omit next line when accountList is dynamic
    private final static int NUM_ACCOUNTS = 3;

    // what the banker can ask of the bank

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

    // what the customer can ask of the bank

    private static final String CUSTOMER_TRANSACTIONS =
    "    Customer transactions: " +
        "deposit, withdraw, transfer, balance, 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;
        // initialize collection:
        accountList   = new BankAccount[NUM_ACCOUNTS]; ///

        /// When accountList is an array, fill it here.
        /// When it's an ArrayList or a TreeMap, delete these lines.
        /// Bank starts with no accounts, banker creates them with
        /// the openNewAccount method.
        accountList[0] = new BankAccount(  0, this);
        accountList[1] = new BankAccount(100, this);
        accountList[2] = new BankAccount(200, this);
    }

    /**
     * Simulates interaction with a Bank.
     * Presents the user with an interactive loop, prompting for
     * banker transactions and in 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("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.
```

```
113
114     private void openNewAccount()
115     {
116         /// when accountList is a dynamic collection
117         /// remove the next two lines, uncomment and complete
118         /// the code between /* and */
119         atm.println(bankName + " is accepting no new customers\n");
120         return;
121
122         /*
123         // prompt for initial deposit
124         int startup = atm.readInt( "Initial deposit: " );
125
126         // create newAccount
127         BankAccount newAccount = new BankAccount( startup, this );
128
129         //  and add it to accountList
130         ???
131
132         // inform user
133         atm.println( "opened new account " + ??? /// name or number
134                         + " with $" + newAccount.getBalance());
135         */
136     }
137
138     // Prompt the customer for transaction to process.
139     // Then send an appropriate message to the account.
140
141     private void processTransactionsForAccount( BankAccount acct )
142     {
143         help( CUSTOMER_TRANSACTIONS );
144
145         String transaction;
146         while (!(transaction =
147                  atm.readWord("    transaction: ")).equals("quit")) {
148
149             if ( transaction.startsWith( "h" ) ) {
150                 help( CUSTOMER_TRANSACTIONS );
151             }
152             else if ( transaction.startsWith( "d" ) ) {
153                 int amount = atm.readInt( "    amount:" );
154                 atm.println("    deposited " + acct.deposit( amount ));
155             }
156             else if ( transaction.startsWith( "w" ) ) {
157                 int amount = atm.readInt( "    amount:" );
158                 atm.println("    withdrew " + acct.withdraw( amount ));
159             }
160             else if (transaction.startsWith("t")) {
161                 atm.print( "    to ");
162                 BankAccount toacct = whichAccount();
163                 if (toacct != null) {
164                     int amount = atm.readInt("    amount to transfer: ");
165                     atm.println("    transfered " +
166                                 toacct.deposit(acct.withdraw(amount)));
167                 }
168             }
```



```
169             else if (transaction.startsWith("b")) {
170                 atm.println("    current balance " +
171                             acct.requestBalance());
172             }
173             else {
174                 atm.println("    sorry, unknown transaction" );
175             }
176         }
177         atm.println();
178     }
179
180     // Prompt for an account name (or number), look it up
181     // in the account list. If it's there, return it;
182     // otherwise report an error and return null.
183
184     private BankAccount whichAccount()
185     {
186         /// prompt for account name or account number
187         /// (whichever is appropriate)
188         int accountNumber = atm.readInt("account number: ");
189
190         /// look up account in accountList
191         /// if it's there, return it
192         /// else the following two lines should execute
193         if ( accountNumber >= 0 && accountNumber < NUM_ACCOUNTS ) {
194             return accountList[accountNumber];
195         }
196         else {
197             atm.println("not a valid account");
198             return null;
199         }
200     }
201
202     // Report bank activity.
203     // For each BankAccount, print the customer id (name or number),
204     // account balance and the number of transactions.
205     // Then print Bank totals.
206
207     private void report()
208     {
209         atm.println( "\nSummaries of individual accounts:" );
210         atm.println( "account  balance   transaction count" );
211         for (int i = 0; i < NUM_ACCOUNTS; i++ ) {             ///
212             atm.println(i + "\t" + accountList[i].getBalance() +     ///
213                         "\t" + accountList[i].getTransactionCount());///
214         }
215
216         atm.println( "\nBank totals");
217         atm.println( "open accounts: " + getNumberOfAccounts() );
218         atm.println( "cash on hand: $" + getBalance());
219         atm.println( "transactions:  " + getTransactionCount());
220         atm.println();
221     }
222
223
224     // Welcome the user to the bank and instruct her on
```

```
225      //  her options.
226
227      private void instructUser()
228      {
229          atm.println( "Welcome to " + bankName );
230          atm.println( "Open some accounts and work with them." );
231          help( BANKER_COMMANDS );
232      }
233
234      // Display a help string.
235
236      private void help( String helpString )
237      {
238          atm.println( helpString );
239          atm.println();
240      }
241
242      /**
243       * Increment bank balance by given amount.
244       *
245       * @param amount the amount increment.
246       */
247
248      public void incrementBalance(int amount)
249      {
250          balance += amount;
251      }
252
253      /**
254       * Increment by one the count of transactions,
255       * for this bank.
256       */
257
258      public void countTransaction()
259      {
260          transactionCount++;
261      }
262
263      /**
264       * Get the number of transactions performed by this bank.
265       *
266       * @return number of transactions performed.
267       */
268
269      public int getTransactionCount( )
270      {
271          return transactionCount ;
272      }
273
274      /**
275       * Get the current bank balance.
276       *
277       * @return current bank balance.
278       */
279
280      public int getBalance()
```



```
281      {
282          return balance;
283      }
284
285      /**
286       * Get the current number of open accounts.
287       *
288       * @return number of open accounts.
289       */
290
291      public int getNumberOfAccounts()
292      {
293          return NUM_ACCOUNTS; /// needs changing ...
294      }
295
296      /**
297       * Run the simulation by creating and then visiting a new Bank.
298       * <p>
299       * A -e argument causes the input to be echoed.
300       * This can be useful for executing the program against
301       * a test script, e.g.,
302       * <pre>
303       *   java Bank -e < Bank.in
304       * </pre>
305       *
306       * @param args the command line arguments:
307       *          <pre>
308       *          -e echo input.
309       *          bankName any other command line argument.
310       *          </pre>
311       */
312
313      public static void main( String[] args )
314      {
315          // parse the command line arguments for the echo
316          // flag and the name of the bank
317
318          boolean echo    = false;        // default does not echo
319          String bankName = "River Bank"; // default bank name
320
321          for (int i = 0; i < args.length; i++ ) {
322              if (args[i].equals("-e")) {
323                  echo = true;
324              }
325              else {
326                  bankName = args[i];
327              }
328          }
329          Bank aBank = new Bank( bankName, new Terminal(echo) );
330          aBank.visit();
331      }
332  }
```

```
// joi/4/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 4
 */

public 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.
     */

    public BankAccount( int initialBalance, Bank issuingBank )
    {
        this.issuingBank = issuingBank;
        deposit( initialBalance );
    }

    /**
     * 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
     */

    public int withdraw( int amount )
    {
        incrementBalance( -amount );
        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
     */

    public int deposit(int amount)
    {
        incrementBalance( amount);
        countTransaction();
        return amount ;
    }

    /**
     * Request for balance. Counts as a transaction.
     *
     * @return current account balance
     */

    public int requestBalance()
    {
        countTransaction();
        return getBalance() ;
    }

    /**
     * Get the current balance.
     * Does NOT count as a transaction.
     *
     * @return current account balance
     */

    public int getBalance()
    {
        return balance;
    }

    /**
     * Increment account balance by given amount.
     * Also increment issuing Bank's balance.
     * Does NOT count as a transaction.
     *
     * @param amount the amount increment.
     */

    public void incrementBalance( int amount )
    {
        balance += amount;
        this.getIssuingBank().incrementBalance( amount );
    }

    /**
     * Get the number of transactions performed by this
     * account. Does NOT count as a transaction.
     *
     * @return number of transactions performed.
     */
```

```
   113
   114      public int getTransactionCount()
   115      {
   116          return transactionCount;
   117      }
   118
   119      /**
   120       * Increment by 1 the count of transactions, for this account
   121       * and for the issuing Bank.
   122       * Does NOT count as a transaction.
   123       */
   124
   125      public void countTransaction()
   126      {
   127          transactionCount++;
   128          this.getIssuingBank().countTransaction();
   129      }
   130
   131      /**
   132       * Get the bank that issued this account.
   133       * Does NOT count as a transaction.
   134       *
   135       * @return issuing bank.
   136       */
   137
   138      public Bank getIssuingBank()
   139      {
   140          return issuingBank;
   141      }
   142  }
```

```
 1  open
 2  1000
 3  open
 4  2000
 5  help
 6  report
 7  open
 8  3000
 9  customer
10  0
11  balance
12  deposit
13  9999
14  balance
15  quit
16  customer
17  1
18  transfer
19  9
20  transfer
21  2
22  45
23  quit
24  exit
```

```
Welcome to River Bank
Open some accounts and work with them.
Banker commands: exit, open, customer, report, help.

banker command: open
Initial deposit: 1000
opened new account 0 with $1000
banker command: open
Initial deposit: 2000
opened new account 1 with $2000
banker command: help
Banker commands: exit, open, customer, report, help.

banker command: report

Summaries of individual accounts:
account  balance   transaction count
0       $1000           1
1       $2000           1

Bank totals
open accounts: 2
cash on hand: $3000
transactions:  2

banker command: open
Initial deposit: 3000
opened new account 2 with $3000
banker command: customer
account number: 0
    Customer transactions: deposit, withdraw, transfer, balance, quit, he

    transaction: balance
    current balance 1000
    transaction: deposit
    amount:9999
    deposited 9999
    transaction: balance
    current balance 10999
    transaction: quit

banker command: customer
account number: 1
    Customer transactions: deposit, withdraw, transfer, balance, quit, he

    transaction: transfer
    to account number: 9
not a valid account
    transaction: transfer
    to account number: 2
    amount to transfer: 45
    transfered 45
    transaction: quit

banker command: exit

```



```
Summaries of individual accounts:
account  balance   transaction count
0       $10999          4
1       $1955           2
2       $3045           2

Bank totals
open accounts: 3
cash on hand: $15999
transactions:  8

Goodbye from River Bank
```

```
 1  open
 2  groucho
 3  1000
 4  customer
 5  harpo
 6  open
 7  harpo
 8  2000
 9  help
10  report
11  open
12  chico
13  3000
14  customer
15  groucho
16  balance
17  deposit
18  9999
19  balance
20  quit
21  customer
22  harpo
23  transfer
24  chico
25  45
26  quit
27  exit
```

```
Welcome to River Bank
Open some accounts and work with them.
Banker commands: exit, open, customer, report, help.

banker command: open
Account name: groucho
Initial deposit: 1000
opened new account groucho with $1000
banker command: customer
account name: harpo
not a valid account
banker command: open
Account name: harpo
Initial deposit: 2000
opened new account harpo with $2000
banker command: help
Banker commands: exit, open, customer, report, help.

banker command: report

Summaries of individual accounts:
account  balance   transaction count
groucho $1000           1
harpo   $2000           1

Bank totals
open accounts: 2
cash on hand: $3000
transactions:  2

banker command: open
Account name: chico
Initial deposit: 3000
opened new account chico with $3000
banker command: customer
account name: groucho
    Customer transactions: deposit, withdraw, transfer, balance, quit, he

    transaction: balance
    current balance 1000
    transaction: deposit
    amount:9999
    deposited 9999
    transaction: balance
    current balance 10999
    transaction: quit

banker command: customer
account name: harpo
    Customer transactions: deposit, withdraw, transfer, balance, quit, he

    transaction: transfer
    to account name: chico
    amount to transfer: 45
    transfered 45
    transaction: quit
```



```

banker command: exit

Summaries of individual accounts:
account  balance   transaction count
chico   $3045           2
groucho $10999          4
harpo   $1955           2

Bank totals
open accounts: 3
cash on hand: $15999
transactions:  8

Goodbye from River Bank
```

```
 1  // joi/examples/Reverse.java
 2  //
 3  //
 4  // Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  import java.util.ArrayList;
 7
 8  /**
 9   * Reverse the order of lines entered from standard input.
10   */
11
12  public class Reverse
13  {
14
15      /**
16       * Read lines typed at the terminal until end-of-file,
17       * saving them in an ArrayList.
18       *
19       * Then print the lines in reverse order.
20       */
21
22      public static void main( String[] args )
23      {
24          Terminal  t    = new Terminal();
25          ArrayList list = new ArrayList();
26          String line;
27
28          while ((line = t.readLine()) != null ) {
29              list.add(line);
30          }
31
32          for (int i = list.size()-1; i >= 0; i--) {
33              line = (String)list.get(i);
34              t.println( line );
35          }
36      }
37  }
```

```
// joi/4/dictionary/Dictionary.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * Model a dictionary with a TreeMap of (word, Definition) pairs.
 *
 * @see Definition
 *
 * @version 4
 */

public class Dictionary
{
    private TreeMap entries;

    /**
     * Construct an empty Dictionary.
     */

    public Dictionary()
    {
        entries = new TreeMap();
    }

    /**
     * Add an entry to this Dictionary.
     *
     * @param word the word being defined.
     * @param definition the Definition of that word.
     */

    public void addEntry( String word, Definition definition )
    {
        entries.put( word, definition );
    }

    /**
     * Look up an entry in this Dictionary.
     *
     * @param word the word whose definition is sought
     * @return the Definition of that word, null if none.
     */

    public Definition getEntry( String word )
    {
        return (Definition)entries.get(word);
    }

    /**
     * Get the size of this Dictionary.
     *
     * @return the number of words.
```



```
     */

    public int getSize()
    {
        return entries.size();
    }

    /**
     * Construct a String representation of this Dictionary.
     *
     * @return a multiline String representation.
     */

    public String toString()
    {
        String str = "";
        String word;
        Definition definition;
        Set allWords = entries.keySet();
        Iterator wordIterator = allWords.iterator();
        while ( wordIterator.hasNext() ) {
            word = (String)wordIterator.next();
            definition = this.getEntry( word );
            str += word + ":\n" + definition.toString() + "\n";
        }
        return str;
    }
}
```

```
 1  // joi/4/dictionary/Definition.java
 2  //
 3  //
 4  // Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  /**
 7   * Model the definition of a word in a dictionary.
 8   *
 9   * @see Dictionary
10   *
11   * @version 4
12   */
13
14  public class Definition
15  {
16      private String definition;    // the defining string
17
18      /**
19       * Construct a simple Definition.
20       *
21       * @param definition the definition.
22       */
23
24      public Definition( String definition )
25      {
26          this.definition = definition;
27      }
28
29      /**
30       * Construct a String representation of this Definition.
31       *
32       * @return the definition string.
33       */
34
35      public String toString()
36      {
37          return definition;
38      }
39  }
```

```
// joi/4/dictionary/Lookup.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * On line word lookup.
 *
* @see Dictionary
 * @see Definition
 *
 * @version 4
 */

public class Lookup
{
    private static Terminal t = new Terminal();
    private static Dictionary dictionary = new Dictionary();

    /**
     * Helper method to fill the dictionary with some simple
     * Definitions.
     *
     * A real Dictionary would live in a file somewhere.
     */

    private static void fillDictionary()
    {
        dictionary.addEntry( "shape",
            new Definition(  "a geometric object in a plane" ) );
        dictionary.addEntry( "quadrilateral",
            new Definition(  "a polygonal shape with four sides" ) );
        dictionary.addEntry( "rectangle",
            new Definition(  "a right-angled quadrilateral" ) );
        dictionary.addEntry( "square",
            new Definition(  "a rectangle having equal sides" ) );
    }

    /**
     * Helper method to print the Definition of a single word,
     * or a message if the word is not in the Dictionary.
     *
     * @param word the word whose definition is wanted.
     */

    private static void printDefinition(String word)
    {
        Definition definition = dictionary.getEntry(word);
        if (definition == null) {
            t.println("sorry, no definition found for " + word);
        }
        else {
            t.println(definition.toString());
        }
    }

```



```
    /**
     * Run the Dictionary lookup.
     *
     * Parse command line arguments for words to look up,
     * "all" prints the whole Dictionary.
     *
     * Then prompt for more words, "quit" to finish.
     *
     * For example,
     * <pre>
     *
     * %> java Lookup shape square circle
     * shape:
     * a geometric object in a plane
     * square:
     * a rectangle having equal sides
     * circle:
     * sorry, no definition found for circle
     *
     * look up words, "quit" to quit
     * word> rectangle
     * a right-angled quadrilateral
     * word> quit
     * %>
     * </pre>
     *
     * @param args the words that we want looked up, supplied as
     *             command line arguments.  If the word "all" is
     *             included, all words are looked up.
     */

    public static void main( String[] args )
    {
        // fill the dictionary (not a big one!)
        fillDictionary();

        // look up some words
        String word;

        // words specified on command line
        for (int i = 0; i < args.length; i++ ) {
            word = args[i];
            if (word.equals("all")) {
                t.println("The whole dictionary (" +
                          dictionary.getSize() + " entries):");
                t.println("-------------------");
                t.println(dictionary.toString());
                t.println("-------------------");
            }
            else {
                t.println(word + ":");
                printDefinition(word);
            }
        }

        // words entered interactively
```

```
   113          t.println("\nlook up words, \"quit\" to quit");
   114          while (true) {
   115              word = t.readWord("word> ");
   116              if (word.equals("quit")) {
   117                  break;
   118              }
   119              printDefinition(word);
   120          }
   121      }
   122  }
```

```
 1  // joi/3/textfiles/TextFile.java
 2  //
 3  //
 4  // Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  import java.util.Date;
 7
 8  /**
 9   * A TextFile mimics the sort of text file that one finds
10   * on a computer's file system.  It has an owner,
11   * a create date (when the file was created),
12   * a modification date (when the file was last modified),
13   * and String contents.
14   *
15   * @version 3
16   */
17
18  public class TextFile
19  {
20      // Private Implementation
21
22      private String owner;       // Who owns the file.
23      private Date   createDate;  // When the file was created.
24      private Date   modDate;     // When the file was last modified.
25      private String contents;    // The text stored in the file.
26
27      // Public Interface
28
29      /**
30       * Construct a new TextFile with given owner and
31       * contents; set the creation and modification dates.
32       *
33       * @param owner the user who owns the file.
34       * @param contents the file's initial contents.
35       */
36
37      public TextFile( String owner, String contents )
38      {
39          this.owner    = owner;
40          this.contents = contents;
41          createDate    = new Date(); // date and time now
42          modDate       = createDate;
43      }
44
45      /**
46       * Replace the contents of the file.
47       *
48       * @param contents the new contents.
49       */
50
51      public void setContents( String contents )
52      {
53          this.contents = contents;
54          modDate = new Date();
55      }
56
```



```
57      /**
58       * The contents of a file.
59       *
60       * @return String contents of the file.
61       */
62
63      public String getContents()
64      {
65          return contents;
66      }
67
68      /**
69       * Append text to the end of the file.
70       *
71       * @param  text the text to be appended.
72       */
73
74      public void append( String text )
75      {
76          this.setContents( contents + text );
77      }
78
79      /**
80       * Append a new line of text to the end of the file.
81       *
82       * @param  text the text to be appended.
83       */
84
85      public void appendLine( String text )
86      {
87          this.setContents(contents + '\n' + text);
88      }
89
90      /**
91       * The size of a file.
92       *
93       * @return the integer size of the file
94       *  (the number of characters in its String contents)
95       */
96
97      public int getSize()
98      {
99          int charCount;
100         charCount = contents.length();
101         return charCount;
102     }
103
104     /**
105      * The data and time of the file's creation.
106      *
107      * @return the file's creation date and time.
108      */
109
110     public String getCreateDate()
111     {
112         return createDate.toString();
```

```
113      }
114
115      /**
116       * The date and time of the file's last modification.
117       *
118       * @return the date and time of the file's last modification.
119       */
120
121      public String getModDate()
122      {
123          return modDate.toString();
124      }
125
126      /**
127       * The file's owner.
128       *
129       * @return the owner of the file.
130       */
131
132      public String getOwner()
133      {
134          return owner;
135      }
136
137      /**
138       * A definition of main(), used only for testing this class.
139       *
140       * Executing
141       * <pre>
142       *     %> java TextFile
143       * </pre>
144       * produces the output:
145       * <pre>
146       * TextFile myTextFile contains 13 characters.
147       * Created by bill, Sat Dec 29 14:02:37 EST 2001
148       * Hello, world.
149       *
150       * append new line "How are you today?"
151       * Hello, world.
152       * How are you today?
153       * TextFile myTextFile contains 32 characters.
154       * Modified Sat Dec 29 14:02:38 EST 2001
155       * </pre>
156       */
157
158      public static void main( String[] args )
159      {
160          Terminal terminal = new Terminal();
161          TextFile myTextFile = new TextFile( "bill", "Hello, world." );
162
163          terminal.println( "TextFile myTextFile contains " +
164                            myTextFile.getSize() + " characters.");
165          terminal.println( "Created by " +  myTextFile.getOwner() +
166                            ", " +
167                            myTextFile.getCreateDate() );
168          terminal.println( myTextFile.getContents() );
```



```
169          terminal.println();
170
171          terminal.println( "append new line \"How are you today?\"" );
172          myTextFile.appendLine( "How are you today?" );
173          terminal.println( myTextFile.getContents() );
174          terminal.println( "TextFile myTextFile contains " +
175                            myTextFile.getSize() + " characters." );
176          terminal.println( "Modified "  +
177                            myTextFile.getModDate() );
178      }
179  }
```

```
// joi/4/textfiles/Directory.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

// This draft contains just stubs for the methods.
// You can invoke them all, but none will do anything.

/**
 * Directory of TextFiles.
 *
 * @version 4
 */

public class Directory
{
    /**
     * Construct a Directory.
     */

    public Directory( )
    {
    }

    /**
     * The size of a directory is the number of TextFiles it contains.
     *
     * @return the number of TextFiles.
     */

    public int getSize()
    {
        return 0;
    }

    /**
     * Add a TextFile to this Directory. Overwrite if a TextFile
     * of that name already exists.
     *
     * @param name the name under which this TextFile is added.
     * @param afile the TextFile to add.
     */

    public void addTextFile(String name, TextFile afile)
    {
    }

    /**
     * Get a TextFile in this Directory, by name .
     *
     * @param filename the name of the TextFile to find.
     * @return the TextFile found, null if none.
     */

    public TextFile retrieveTextFile( String filename )
    {
```



```
        return null;
    }

    /**
     * Get the contents of this Directory as an array of
     * the file names, each of which is a String.
     *
     * @return the array of names.
     */

    public String[] getFileNames()
    {
        // pseudocode for an implementation:
        // declare an array of String
        // create that array with as many spaces as there
        //   are TextFile's in this Directory
        // loop through the keys of the TreeMap of TextFiles,
        // adding each String key to the array
        // return the array

        // the next line is there because we have to return
        // _something_ in order to satisfy the compiler
        return new String[0];
    }

    /**
     *  main, for unit testing.
     *
     * The command
     * <pre>
     *   java Directory
     * </pre>
     * should produce output
     * <pre>
     * bill     17      Sun Jan 06 19:40:13 EST 2003    diary
     * eb       12      Sun Jan 06 19:40:13 EST 2003    greeting
     * </pre>
     * (with current dates, of course).
     */

    public static void main( String[] args )
    {
        Directory dir = new Directory();
        dir.addTextFile("greeting", new TextFile("eb", "Hello, world"));
        dir.addTextFile("diary", new TextFile("bill", "Writing Directory"
        // now list TextFiles in dir to get output specified
    }
}
```

```
// joi/4/estore/EStore.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * An EStore object simulates the behavior of a simple on line
 * shopping web site.
 *
 * It contains a Terminal object to model the customer's browser
 * and a Catalog of Items that may be purchased and
 * then added to the customer's shoppingCart.
 *
 * @version 4
 */

public class EStore
{
    private String   storeName;
    private Terminal browser;
    private Catalog  catalog;

    /**
     *  Construct a new EStore.
     *
     *  @param storeName the name of the EStore
     *  @param browser the visitor's Terminal.
     */

    public EStore( String storeName, Terminal browser )
    {
        this.browser   = browser;
        this.storeName = storeName;
        this.catalog   = new Catalog();
        catalog.addItem( new Item("quaffle", 55) );
        catalog.addItem( new Item("bludger", 15) );
        catalog.addItem( new Item("snitch",  1000) );
    }

    /**
     *  Visit this EStore.
     *
     *  Execution starts here when the store opens for
     *  business. User can visit as a customer, act as
     *  the manager, or exit.
     */

    public void visit()
    {
        // Print a friendly welcome message.
        browser.println( "Welcome to " + storeName );
        while (true) { // an infinite loop ...
            browser.println();
            String whoAreYou = browser.readWord(
               storeName + " (manager, visit, exit): ");
            if (whoAreYou.equals("exit")) {
```



```
                break; // leave the while loop
            }
            if (whoAreYou.equals("manager")) {
                managerVisit();
            }
            if (whoAreYou.equals("visit")) {
                customerVisit();
            }
        }
    }

    /**
     *  Manager options:
     *
     *  examine the catalog
     *  add an Item to the catalog
     *  quit
     */
    private void managerVisit( )
    {
        while (true) {
            String cmd =
              browser.readWord("manager command (show, new, quit):");
            if (cmd.equals("quit")) {
                break;  // leave manager command while loop
            }
            else if (cmd.equals("show")) {
                catalog.show(browser);
            }
            else if (cmd.equals("new")) {
                String itemName = browser.readWord("  item name:     ");
                int cost        = browser.readInt( "  cost:          ");
                catalog.addItem( new Item(itemName, cost) );
            }
            else {
                browser.println("unknown manager command: " + cmd);
            }
        }
    }

    /**
     * Customer visits this EStore.
     *
     * Loop allowing customer to select items to add to her
     * shoppingCart.
     */

    private void customerVisit( )
    {
        // Create a new, empty ShoppingCart.
        ShoppingCart basket = new ShoppingCart();

        browser.println( "Currently available:");
        catalog.show(browser);
        while ( true ) { // loop forever ...
            String nextPurchase = browser.readWord(
```

```
113                     "select your purchase, checkout, help: ");
114
115             if ( nextPurchase.equals("checkout" )) break; // leave loop!
116
117             if ( nextPurchase.equals("help" )) {
118                 catalog.show(browser);
119                 continue; // go back to top of while loop
120             }
121             // customer has entered the name of an Item
122             basket.addItem( catalog.getItem(nextPurchase) );
123         }
124
125         int numberPurchased = basket.getCount();
126         browser.println("We are shipping these " +
127                         basket.getCount() + " Items:");
128         basket.showContents(browser);
129         browser.println("and charging your account $" + basket.getCost())
130         browser.println("Thank you for shopping at " + storeName);
131     }
132
133     /**
134      *  The EStore simulation program begins here when the user
135      *  issues the command <code>java EStore</code>
136      *
137      *  If first command line argument is "-e" instantiate a
138      *  Terminal that echoes its input.
139      *
140      *  The next command line argument (if there is one)
141      *  is the name of the EStore.
142      *
143      *  @param args <-e> <storeName>
144      */
145     public static void main( String[] args )
146     {
147
148         String storeName = "Virtual Minimal Minimall"; //default
149
150         // check to see if first argument is "-e"
151         boolean echo = ( (args.length > 0) && (args[0].equals("-e")) );
152
153         // if first argument was "-e" then look at second for store name
154         int nextArg = (echo ? 1 : 0 );
155
156         if (args.length > nextArg) {
157             storeName = args[nextArg];
158         }
159
160         // Print this to simulate internet search.
161         System.out.println("connecting ...");
162
163         // Create an EStore object and visit it
164         (new EStore(storeName, new Terminal(echo))).visit();
165     }
166 }
```

```
 1  // joi/4/estore/ShoppingCart.java
 2  //
 3  //
 4  // Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  /**
 7   * A ShoppingCart keeps track of a customer's purchases.
 8   *
 9   * @see EStore
10   * @version 4
11   */
12
13  public class ShoppingCart
14  {
15      /// replace these two fields by a single ArrayList
16      private int count; // number of Items in this ShoppingCart
17      private int cost;  // total cost of Items in this ShoppingCart
18
19      /**
20       * Construct a new empty ShoppingCart.
21       */
22
23      public ShoppingCart()
24      {
25          count = 0;
26          cost  = 0;
27      }
28
29      /**
30       * Add an Item to this ShoppingCart.
31       *
32       * @param item the Item to add.
33       */
34
35      public void addItem( Item item )
36      {
37          /// this code just keeps track of the totals
38          /// replace it with code that adds the item to the list
39          count++;
40          this.cost += item.getCost(); // Java idiom: a += b means a = a +
41      }
42
43      /**
44       * Return an Item from this ShoppingCart.
45       *
46       * @param item the Item to return.
47       */
48
49      public void returnItem( Item item )
50      {
51          /// look through the list looking for Item
52          /// remove it if it's there
53      }
54
55      /**
56       * What happens when this ShoppingCart is asked how many
```



```
57       * Items it contains.
58       *
59       * @return the number of items in this ShoppingCart.
60       */
61
62      public int getCount()
63      {
64          /// get this information from the list,
65          /// since the count field no longer exists
66          return count;
67      }
68
69      /**
70       * What happens when this ShoppingCart is asked the total
71       * cost of the Items it contains.
72       *
73       * @return the total cost of the items in this ShoppingCart.
74       */
75      public int getCost()
76      {
77          /// get this information from the list,
78          /// since the cost field no longer exists
79          return cost;
80      }
81
82      /**
83       * Write the contents of this ShoppingCart to a Terminal.
84       *
85       * @param t the Terminal to use for output.
86       */
87
88      public void showContents( Terminal t)
89      {
90          /// work to do here ...
91          t.println("  [sorry, can't yet print ShoppingCart contents]");
92      }
93  }
```

```
 1  // joi/4/estore/Item..java
 2  //
 3  //
 4  // Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  /**
 7   * An Item models an object that might be stocked in a store.
 8   * Each Item has a cost.
 9   *
10   * @version 4
11   */
12
13  public class Item
14  {
15      private int cost;
16      private String name;
17
18      /**
19       *  Construct an Item object.
20       *
21       *  @param name the nme of this Item.
22       *  @param cost the cost of this Item.
23       */
24
25      public Item( String name, int cost )
26      {
27          this.name = name;
28          this.cost = cost;
29      }
30
31      /**
32       * How much does this Item cost?
33       *
34       * @return the cost.
35       */
36
37      public int getCost()
38      {
39          return cost;
40      }
41
42      /**
43       * What is this Item called?
44       *
45       * @return the name.
46       */
47
48      public String getName()
49      {
50          return name;
51      }
52  }
```

```
 1  // joi/4/estore/Catalog.java
 2  //
 3  //
 4  // Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  import java.util.TreeMap;
 7
 8  /**
 9   * A Catalog models the collection of Items that an
10   * EStore might carry.
11   *
12   * @see EStore
13   *
14   * @version 4
15   */
16
17  public class Catalog
18  {
19      private TreeMap items;
20
21      /**
22       *  Construct a Catalog object.
23       */
24
25      public Catalog( )
26      {
27          items = new TreeMap();
28      }
29
30      /**
31       * Add an Item to this Catalog.
32       *
33       * @param item the Item to add.
34       */
35
36      public void addItem( Item item )
37      {
38          items.put( item.getName(), item );
39      }
40
41      /**
42       * Get an Item from this Catalog.
43       *
44       * @param itemName the name of the wanted Item
45       *
46       * @return the Item, null if none.
47       */
48
49      public Item getItem( String itemName )
50      {
51          return (Item)items.get(itemName);
52      }
53
54      /**
55       * Display the contents of this Catalog.
56       *
```



```
57       * @param t the Terminal to print to.
58       */
59
60      public void show( Terminal t )
61      {
62          // loop on items, printing name and cost
63          t.println("  [sorry, can't yet print Catalog contents]");
64      }
65  }
```