```
 1  // joi/7/bank/Bank.java
 2  //
 3  //
 4  // Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  import java.util.*;
 7
 8  /**
 9   * A Bank object simulates the behavior of a simple bank/ATM.
10   * It contains a Terminal object and a collection of
11   * BankAccount objects.
12   *
13   * The visit method opens this Bank for business,
14   * prompting the customer for input.
15   *
16   * To create a Bank and open it for business issue the command
17   * <code>java Bank</code>.
18   *
19   * @see BankAccount
20   * @version 7
21   */
22
23  public class Bank
24  {
25      private String bankName;           // the name of this Bank
26      private Terminal atm;              // for talking with the customer
27      private int balance = 0;           // total cash on hand
28      private int transactionCount = 0;  // number of Bank transactions
29      private Month month;               // the current month.
30      private Map accountList;           // mapping names to accounts.
31
32      private int checkFee = 2;            // cost for each check
33      private int transactionFee = 1;      // fee for each transaction
34      private int monthlyCharge = 5;       // monthly charge
35      private double interestRate = 0.05;  // annual rate paid on savings
36      private int maxFreeTransactions = 3; // for savings accounts
37
38      // what the banker can ask of the bank
39
40      private static final String BANKER_COMMANDS =
41      "Banker commands: " +
42          "exit, open, customer, nextmonth, report, help.";
43
44      // what the customer can ask of the bank
45
46      private static final String CUSTOMER_TRANSACTIONS =
47      "    Customer transactions: " +
48      "deposit, withdraw, transfer, balance, cash check, quit, help.";
49
50      /**
51       * Construct a Bank with the given name and Terminal.
52       *
53       * @param bankName the name for this Bank.
54       * @param atm  this Bank's Terminal.
55       */
56
```



```
57      public Bank( String bankName, Terminal atm )
58      {
59          this.atm      = atm;
60          this.bankName = bankName;
61          accountList   = new TreeMap();
62          month         = new Month();
63      }
64
65      /**
66       * Simulates interaction with a Bank.
67       * Presents the user with an interactive loop, prompting for
68       * banker transactions and in the case of the banker
69       * transaction "customer", an account id and further
70       * customer transactions.
71       */
72
73      public void visit()
74      {
75          instructUser();
76
77          String command;
78          while (!(command =
79                   atm.readWord("banker command: ")).equals("exit")) {
80
81              if (command.startsWith("h")) {
82                  help( BANKER_COMMANDS );
83              }
84              else if (command.startsWith("o")) {
85                  openNewAccount();
86              }
87              else if (command.startsWith("n")) {
88                  newMonth();
89              }
90              else if (command.startsWith("r")) {
91                  report();
92              }
93              else if (command.startsWith( "c" ) ) {
94                  BankAccount acct = whichAccount();
95                  if ( acct != null ) {
96                          processTransactionsForAccount( acct );
97                  }
98              }
99              else {
100                 // Unrecognized Request
101                 atm.println( "unknown command: " + command );
102             }
103         }
104         report();
105         atm.println( "Goodbye from " + bankName );
106     }
107
108
109     // Open a new bank account,
110     // prompting the user for information.
111
112     private void openNewAccount()
```

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



```
169                  }
170                  else if ( transaction.startsWith( "w" ) ) {
171                      int amount = readPosAmt( "    amount:" );
172                      atm.println("    withdrew "
173                                  + acct.withdraw( amount ));
174                  }
175                  else if ( transaction.startsWith( "c" ) ) {
176                      int amount = readPosAmt( "    amount of check: " );
177                      try { // to cast acct to CheckingAccount ...
178                          atm.println("    cashed check for " +
179                             ((CheckingAccount) acct).honorCheck( amount ))
180                      }
181                      catch (ClassCastException e) {
182                          // if not a checking account, report error
183                          atm.errPrintln(
184                              "    Sorry, not a checking account." );
185                      }
186                  }
187                  else if (transaction.startsWith("t")) {
188                      atm.print( "    to ");
189                      BankAccount toacct = whichAccount();
190                      if (toacct != null) {
191                      int amount = readPosAmt("    amount to transfer: ");
192                      atm.println("    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   }
```

```
 1  // joi/7/bank/BankAccount.java
 2  //
 3  //
 4  // Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  /**
 7   * A BankAccount object has private fields to keep track
 8   * of its current balance, the number of transactions
 9   * performed and the Bank in which it is an account, and
10   * and public methods to access those fields appropriately.
11   *
12   * @see Bank
13   * @version 7
14   */
15
16  public abstract class BankAccount
17  {
18      private int  balance = 0;          // Account balance (whole dollars)
19      private int  transactionCount = 0; // Number of transactions performe
20      private Bank issuingBank;          // Bank issuing this account
21
22      /**
23       * Construct a BankAccount with the given initial balance and
24       * issuing Bank. Construction counts as this BankAccount's
25       * first transaction.
26       *
27       * @param initialBalance the opening balance.
28       * @param issuingBank the bank that issued this account.
29       *
30       * @exception InsufficientFundsException when appropriate.
31       */
32
33      protected BankAccount( int initialBalance, Bank issuingBank )
34          throws InsufficientFundsException
35      {
36          this.issuingBank = issuingBank;
37          deposit( initialBalance );
38      }
39
40      /**
41       * Get transaction fee.  By default, 0.
42       * Override this for accounts having transaction fees.
43       *
44       * @return the fee.
45       */
46
47      protected int getTransactionFee()
48      {
49          return 0;
50      }
51
52      /**
53       * The bank that issued this account.
54       *
55       * @return the Bank.
56       */
```



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

```
113          countTransaction();
114          return getBalance() ;
115      }
116
117      /**
118       * Get the current balance.
119       * Does NOT count as a transaction.
120       *
121       * @return current account balance
122       */
123
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  }
```

```
// joi/7/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 7
 */

public class Month
{
    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 );
    }
}
```

```
 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  }
```

```
// joi/7/juno/Juno.java
//
//
//  Copyright 2003 Bill Campbell and Ethan Bolker

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

/**
 * Juno (Juno's Unix NOt) mimics a command line operating system
 * like Unix.
 * <p>
 * A Juno system has a name, a set of Users, a JFile system,
 * a login process and a set of shell commands.
 *
 * @see User
 * @see JFile
 * @see ShellCommand
 *
 * @version 7
 */

public class Juno
{
    private final static String os      = "Juno";
    private final static String version = "7";

    private String    hostName;   // host machine name
    private Map       users;      // lookup table for Users
    private Terminal  console;    // for input and output

    private Directory slash;      // root of JFile system
    private Directory userHomes;  // for home directories

    private ShellCommandTable commandTable; // shell commands

    /**
     * Construct a Juno (operating system) object.
     *
     * @param hostName  the name of the host on which it's running.
     * @param echoInput should all input be echoed as output?
     */

    public Juno( String hostName, boolean echoInput )
    {
        // initialize the Juno environment ...

        this.hostName = hostName;
        console       = new Terminal( echoInput );
        users         = new TreeMap();           // for registered Users
        commandTable  = new ShellCommandTable(); // for shell commands

        // the file system

        slash     = new Directory( "", null, null );
```



```
        User root = new User( "root", slash, "Rick Martin" );
        users.put( "root", root );
        slash.setOwner(root);
        userHomes = new Directory( "users", root, slash );

        // create, then start a command line login interpreter

        LoginInterpreter interpreter
            = new LoginInterpreter( this, console );
        interpreter.CLILogin();
    }

    /**
     * The name of the host computer on which this system
     * is running.
     *
     * @return the host computer name.
     */

    public String getHostName()
    {
        return hostName;
    }

    /**
     * The name of this operating system.
     *
     * @return the operating system name.
     */

    public String getOS()
    {
        return os;
    }

    /**
     * The version number for this system.
     *
     * @return the version number.
     */

    public String getVersion()
    {
        return version;
    }

    /**
     * The directory containing all user homes for this system.
     *
     * @return the directory containing user homes.
     */

    public Directory getUserHomes()
    {
        return userHomes;
    }
```

```
113
114     /**
115      * The shell command table for this system.
116      *
117      * @return the shell command table.
118      */
119
120     public ShellCommandTable getCommandTable()
121     {
122         return commandTable;
123     }
124
125     /**
126      * Look up a user by user name.
127      *
128      * @param username the user's name.
129      * @return the appropriate User object.
130      */
131
132     public User lookupUser( String username )
133     {
134         return (User) users.get( username );
135     }
136
137     /**
138      * Create a new User.
139      *
140      * @param userName the User's login name.
141      * @param home her home Directory.
142      * @param realName her real name.
143      * @return newly created User.
144      */
145
146     public User createUser( String userName, Directory home,
147                             String realName )
148     {
149         User newUser = new User( userName, home, realName );
150         users.put( userName, newUser );
151         return newUser;
152     }
153
154     /**
155      * The Juno system may be given the following command line
156      * arguments.
157      * <pre>
158      *
159      * -e:         Echo all input (useful for testing).
160      *
161      * -version:   Report the version number and exit.
162      *
163      * [hostname]: The name of the host on which
164      *             Juno is running (optional).
165      * </pre>
166      */
167
168     public static void main( String[] args )
```



```
169     {
170         // Parse command line options
171
172         boolean echoInput = false;
173         String  hostName  = "mars";
174
175         for (int i=0; i < args.length; i++) {
176             if (args[i].equals("-version")) {
177                 System.out.println( os + " version " + version );
178                 System.exit(0);
179             }
180             if (args[i].equals("-e")) {
181                 echoInput = true;
182             }
183             else {
184                 hostName = args[i];
185             }
186         }
187
188         // create a Juno instance, which will start itself
189
190         new Juno( hostName, echoInput );
191     }
192 }
```

```
// joi/7/juno/LoginInterpreter.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Interpreter for Juno login commands.
 *
 * There are so few commands that if-then-else logic is OK.
 *
 * @version 7
 */

public class LoginInterpreter
{
    private static final String LOGIN_COMMANDS =
        "help, register, <username>, exit";

    private Juno     system;  // the Juno object
    private Terminal console; // for i/o

    /**
     * Construct a new LoginInterpreter for interpreting
     * login commands.
     *
     * @param system the system creating this interpreter.
     * @param console the Terminal used for input and output.
     */

    public LoginInterpreter( Juno system, Terminal console)
    {
        this.system  = system;
        this.console = console;
    }

    /**
     * Set the console for this interpreter.  Used by the
     * creator of this interpreter.
     *
     * @param console the Terminal to be used for input and output.
     */

    public void setConsole( Terminal console )
    {
        this.console = console;
    }

    /**
     * Simulates behavior at login: prompt.
     * CLI stands for "Command Line Interface".
     */

    public void CLILogin()
    {
```



```
        welcome();
        boolean moreWork = true;
        while( moreWork ) {
            moreWork = interpret( console.readLine( "Juno login: " ) );
        }
    }

    // Parse user's command line and dispatch appropriate
    // semantic action.
    //
    // return true unless "exit" command or null inputLine.

    private boolean interpret( String inputLine )
    {
        if (inputLine == null) return false;
        StringTokenizer st =
            new StringTokenizer( inputLine );
        if (st.countTokens() == 0) {
            return true; // skip blank line
        }
        String visitor = st.nextToken();
        if (visitor.equals( "exit" )) {
            return false;
        }
        if (visitor.equals( "register" )) {
            register( st );
        }
        else if (visitor.equals( "help" )) {
            help();
        }
        else {
            User user = system.lookupUser(visitor);
            new Shell( system, user, console );
        }
        return true;
    }

    // Register a new user, giving him or her a login name and a
    // home directory on the system.
    //
    // StringTokenizer argument contains the new user's login name
    // followed by full real name.

    private void register( StringTokenizer st )
    {
        String userName = st.nextToken();
        String realName = st.nextToken("").trim();
        Directory home  = new Directory( userName, null,
                                         system.getUserHomes() );
        User user = system.createUser( userName, home, realName );
        home.setOwner( user );
    }

    // Display a short welcoming message, and remind users of
    // available commands.

```

```
   113      private void welcome()
   114      {
   115          console.println( "Welcome to " + system.getHostName() +
   116                           " running " + system.getOS() +
   117                           " version " + system.getVersion() );
   118          help();
   119      }
   120
   121      // Remind user of available commands.
   122
   123      private void help()
   124      {
   125          console.println( LOGIN_COMMANDS );
   126          console.println("");
   127      }
   128  }
```

```
// joi/7/juno/Shell.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * Models a shell (command interpreter)
 *
 * The Shell knows the (Juno) system it's working in,
 * the User who started it,
 * and the console to which to send output.
 *
 * It keeps track of the the current working directory (.) .
 *
 * @version 7
 */

public class Shell
{
    private Juno system;       // the operating system object
    private User user;         // the user logged in
    private Terminal console;  // the console for this shell
    private Directory dot;     // the current working directory

    /**
     * Construct a login shell for the given user and console.
     *
     * @param system a reference to the Juno system.
     * @param user the User logging in.
     * @param console a Terminal for input and output.
     */

    public Shell( Juno system, User user, Terminal console )
    {
        this.system  = system;
        this.user    = user;
        this.console = console;
        dot          = user.getHome(); // default current directory
        CLIShell();
    }

    // Run the command line interpreter

    private void CLIShell()
    {
        boolean moreWork = true;
        while(moreWork) {
            moreWork = interpret( console.readLine( getPrompt() ) );
        }
        console.println("goodbye");
    }

    // Interpret a String of the form
    //     shellcommand command-arguments
```



```
    //
    // return true, unless shell command is logout.

    private boolean interpret( String inputLine )
    {
        StringTokenizer st = stripComments(inputLine);
        if (st.countTokens() == 0) {           // skip blank line
            return true;
        }
        String commandName = st.nextToken();
        ShellCommand commandObject =
            system.getCommandTable().lookup( commandName );
        if (commandObject == null ) {                              // EEE
            console.errPrintln("Unknown command: " + commandName);// EEE
            return true;
        }                                                          // EEE
        try {                                                      // EEE
            commandObject.doIt( st, this );                        // EEE
        }                                                          // EEE
        catch (ExitShellException e) {
            return false;
        }
        catch (BadShellCommandException e) {                       // EEE
            console.errPrintln( "Usage: " + commandName + " " +  // EEE
                                e.getCommand().getArgString() ); // EEE
        }                                                          // EEE
        catch (JunoException e) {                                  // EEE
            console.errPrintln( e.getMessage() );                  // EEE
        }                                                          // EEE
        catch (Exception e) {                                      // EEE
            console.errPrintln( "you should never get here" );   // EEE
            console.errPrintln( e.toString() );                    // EEE
        }                                                          // EEE
        return true;
    }

    // Strip characters from '#' to end of line, create and
    // return a StringTokenizer for what's left.

    private StringTokenizer stripComments( String line )
    {
        int commentIndex = line.indexOf('#');
        if (commentIndex >= 0) {
            line = line.substring(0,commentIndex);
        }
        return new StringTokenizer(line);
    }

    /**
     * The prompt for the CLI.
     *
     * @return the prompt string.
     */

    public String getPrompt()
    {
```

```
113           return system.getHostName() + "> ";
114       }
115
116       /**
117        * The User associated with this shell.
118        *
119        * @return the user.
120        */
121
122       public User getUser()
123       {
124           return user;
125       }
126
127       /**
128        * The current working directory for this shell.
129        *
130        * @return the current working directory.
131        */
132
133       public Directory getDot()
134       {
135           return dot;
136       }
137
138       /**
139        * Set the current working directory for this Shell.
140        *
141        * @param dot the new working directory.
142        */
143
144       public void setDot(Directory dot)
145       {
146           this.dot = dot;
147       }
148
149       /**
150        * The console associated with this Shell.
151        *
152        * @return the console.
153        */
154
155       public Terminal getConsole()
156       {
157           return console;
158       }
159
160       /**
161        * The Juno object associated with this Shell.
162        *
163        * @return the Juno instance that created this Shell.
164        */
165
166       public Juno getSystem()
167       {
168           return system;
```



```
169       }
170   }
```

```
 1  // joi/7/juno/ShellCommand.java
 2  //
 3  //
 4  // Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  import java.util.*;
 7
 8  /**
 9   * Model those features common to all ShellCommands.
10   *
11   * Each concrete extension of this class provides a constructor
12   * and an implementation for method doIt.
13   *
14   * @version 7
15   */
16
17  public abstract class ShellCommand
18  {
19      private String helpString;   // documents the command
20      private String argString;    // any args to the command
21
22      /**
23       * A constructor, always called (as super()) by the subclass.
24       * Used only for commands that have arguments.
25       *
26       * @param helpString a brief description of what the command does.
27       * @param argString a prototype illustrating the required arguments.
28       */
29
30      protected ShellCommand( String helpString, String argString )
31      {
32          this.argString  = argString;
33          this.helpString = helpString;
34      }
35
36      /**
37       * A constructor for commands having no arguments.
38       *
39       * @param helpString a brief description of what the command does.
40       */
41
42      protected ShellCommand( String helpString )
43      {
44          this( helpString, "" );
45      }
46
47      /**
48       * Execute the command.
49       *
50       * @param args the remainder of the command line.
51       * @param sh   the current shell
52       *
53       * @exception JunoException for reporting errors
54       */
55
56      public abstract void doIt( StringTokenizer args, Shell sh )
```



```
57          throws JunoException;
58
59      /**
60       * Help for this command.
61       *
62       * @return the help string.
63       */
64
65      public String getHelpString()
66      {
67          return helpString;
68      }
69
70      /**
71       * The argument string prototype.
72       *
73       * @return the argument string prototype.
74       */
75
76      public String getArgString()
77      {
78          return argString;
79      }
80  }
```

```
// joi/7/juno/ShellCommandTable.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * A ShellCommandTable object maintains a dispatch table of
 * ShellCommand objects keyed by the command names used to invoke
 * them.
 *
 * To add a new shell command to the table, install it from
 * method fillTable().
 *
 * @see ShellCommand
 *
 * @version 7
 */

public class ShellCommandTable
{
    private Map table = new TreeMap();

    /**
     * Construct and fill a shell command table.
     */

    public ShellCommandTable()
    {
        fillTable();
    }

    /**
     * Get a ShellCommand, given the command name key.
     *
     * @param key the name associated with the command we're
     *            looking for.
     *
     * @return the command we're looking for, null if none.
     */

    public ShellCommand lookup( String key )
    {
        ShellCommand commandObject = (ShellCommand) table.get( key );
        if (commandObject != null) {
            return commandObject;
        }

        // try to load dynamically
        // construct classname = "KeyCommand"
        char[] chars = (key + "Command").toCharArray();
        chars[0] = key.toUpperCase().charAt(0);
        String classname = new String(chars);
        try {
            commandObject =
```



```
                (ShellCommand)Class.forName(classname).newInstance();
        }
        catch (Exception e) { // couldn't find class
            return null;
        }
        install(key, commandObject); // put it in table for next time
        return commandObject;
    }

    /**
     * Get an array of the command names.
     *
     * @return the array of command names.
     */

    public String[] getCommandNames()
    {
        return (String[]) table.keySet().toArray( new String[0] );
    }

    // Associate a command name with a ShellCommand.

    private void install( String commandName, ShellCommand command )
    {
        table.put( commandName, command );
    }

    // Fill the dispatch table with ShellCommands, keyed by their
    // command names.

    private void fillTable()
    {
        install( "list", new ListCommand() );
        install( "cd", new CdCommand() );
        install( "newfile", new NewfileCommand() );
        install( "remove", new RemoveCommand() );
        install( "help", new HelpCommand() );
        install( "mkdir", new MkdirCommand() );
        install( "type", new TypeCommand() );
        install( "logout", new LogoutCommand() );
    }
}
```

```
 1  // joi/7/juno/MkdirCommand.java
 2  //
 3  //
 4  // Copyright 2003, Bill Campbell and Ethan Bolker
 5
 6  import java.util.*;
 7
 8  /**
 9   * The Juno shell command to create a new directory.
10   * Usage:
11   * <pre>
12   *     mkdir directory-name
13   * </pre>
14   *
15   * @version 7
16   */
17
18  public class MkdirCommand extends ShellCommand
19  {
20      MkdirCommand()
21      {
22          super( "create a subdirectory of the current directory",
23                 "directory-name" );
24      }
25
26      /**
27       * Create a new Directory in the current Directory.
28       *
29       * @param args the remainder of the command line.
30       * @param sh the current shell.
31       *
32       * @exception JunoException for reporting errors.
33       */
34
35      public void doIt( StringTokenizer args, Shell sh )
36           throws JunoException
37      {
38          String filename = args.nextToken();
39          new Directory( filename, sh.getUser(), sh.getDot() );
40      }
41  }
```

```
// joi/7/juno/TypeCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to display the contents of a
 * text file.
 * Usage:
 * <pre>
 *     type textfile
 * </pre>
 *
 * @version 7
 */

public class TypeCommand extends ShellCommand
{
    TypeCommand()
    {
        super( "display contents of a TextFile", "textfile" );
    }

    /**
     * Display the contents of a TextFile.
     *
     * @param args the remainder of the command line.
     * @param sh the current Shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
        throws JunoException                                // EEE
    {
        String filename;
        try {                                               // EEE
            filename = args.nextToken();
        }                                                   // EEE
        catch (NoSuchElementException e) {
            throw new BadShellCommandException( this );     // EEE
        }                                                   // EEE
        try {                                               // EEE
            sh.getConsole().println(
                ( (TextFile) sh.getDot().
                  retrieveJFile( filename ) ).getContents() );
        }                                                   // EEE
        catch (NullPointerException e) {                    // EEE
            throw new JunoException( "JFile does not exist: "  // EEE
                                   + filename);               // EEE
        }                                                     // EEE
        catch (ClassCastException e) {                        // EEE
            throw new JunoException( "JFile not a text file: " // EEE
                                   + filename);               // EEE
        }                                                     // EEE
    }                                                         // EEE
}
```

```
 1  // joi/7/juno/HelpCommand.java
 2  //
 3  //
 4  // Copyright 2003, Bill Campbell and Ethan Bolker
 5
 6  import java.util.*;
 7
 8  /**
 9   * The Juno shell command to display help on the shell commands.
10   * Usage:
11   * <pre>
12   *     help
13   * </pre>
14   *
15   * @version 7
16   */
17
18  public class HelpCommand extends ShellCommand
19  {
20      HelpCommand()
21      {
22          super( "display ShellCommands" );
23      }
24
25      /**
26       * Print out help for all commands.
27       *
28       * @param args the remainder of the command line.
29       * @param sh   the current shell
30       *
31       * @exception JunoException for reporting errors
32       */
33
34      public void doIt( StringTokenizer args, Shell sh )
35           throws JunoException
36      {
37          // Get command keys from global table, print them out.
38
39          sh.getConsole().println( "shell commands" );
40          ShellCommandTable table = sh.getSystem().getCommandTable();
41          String[] names = table.getCommandNames();
42          for (int i = 0; i < names.length; i++ ) {
43              String cmdname = names[i];
44              ShellCommand cmd =
45                  (ShellCommand) table.lookup( cmdname );
46              sh.getConsole().
47                  println( "  " + cmdname + ": " + cmd.getHelpString() );
48          }
49      }
50  }
```

```
 1  // joi/7/juno/NewfileCommand.java
 2  //
 3  //
 4  // Copyright 2003, Bill Campbell and Ethan Bolker
 5
 6  import java.util.*;
 7
 8  /**
 9   * The Juno shell command to create a text file.
10   * Usage:
11   * <pre>
12   *     newfile filename contents
13   * </pre>
14   *
15   * @version 7
16   */
17
18  public class NewfileCommand extends ShellCommand
19  {
20      NewfileCommand()
21      {
22          super( "create a new TextFile", "filename contents" );
23      }
24
25      /**
26       * Create a new TextFile in the current Directory.
27       *
28       * @param args the remainder of the command line.
29       * @param sh the current shell.
30       *
31       * @exception JunoException for reporting errors
32       */
33
34      public void doIt( StringTokenizer args, Shell sh )
35           throws JunoException
36      {
37          String filename;
38          String contents;
39          filename = args.nextToken();
40          contents  = args.nextToken("").trim(); // rest of line
41          new TextFile( filename, sh.getUser(),
42                        sh.getDot(), contents );
43      }
44  }
```

```
// joi/7/juno/CdCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to change directory.
 * Usage:
 * <pre>
 *     cd [directory]
 * </pre>
 * for moving to the named directory.
 *
 * @version 7
 */

class CdCommand extends ShellCommand
{
    CdCommand()
    {
        super( "change current directory", "[ directory ]" );
    }

    /**
     * Move to the named directory
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        String dirname = "";
        Directory d = sh.getUser().getHome(); // default
        if ( args.hasMoreTokens() ) {
            dirname = args.nextToken();
            if (dirname.equals("..")) {
                if (sh.getDot().isRoot())
                    d = sh.getDot(); // no change
                else
                    d = sh.getDot().getParent();
            }
            else if (dirname.equals(".")) {
                d = sh.getDot(); // no change
            }
            else {
                d = (Directory)(sh.getDot().retrieveJFile(dirname));
            }
        }
        sh.setDot( d );
    }
```



```
}
```

```
 1  // joi/7/juno/ListCommand.java
 2  //
 3  //
 4  // Copyright 2003, Bill Campbell and Ethan Bolker
 5
 6  import java.util.*;
 7
 8  /**
 9   * The Juno shell command to list contents of the current directory.
10   * Usage:
11   * <pre>
12   *     list
13   * </pre>
14   *
15   * @version 7
16   */
17
18  public class ListCommand extends ShellCommand
19  {
20      // The constructor adds this object to the global table.
21
22      ListCommand()
23      {
24          super( "list contents of current directory" );
25      }
26
27      /**
28       * List contents of the current working directory.
29       *
30       * @param args the remainder of the command line.
31       * @param sh   the current shell
32       *
33       * @exception JunoException for reporting errors
34       */
35
36      public void doIt( StringTokenizer args, Shell sh )
37          throws JunoException
38      {
39          Terminal terminal = sh.getConsole();
40          Directory dir     = sh.getDot();
41          String[] fileNames = dir.getFileNames();
42
43          terminal.println( dir.getPathName() );
44          for ( int i = 0; i < fileNames.length; i++ ) {
45              String fileName = fileNames[i];
46              JFile  jfile    = dir.retrieveJFile( fileName );
47              terminal.println( jfile.toString() );
48          }
49      }
50  }
```

```
 1  // joi/7/juno/LogoutCommand.java
 2  //
 3  //
 4  // Copyright 2003, Bill Campbell and Ethan Bolker
 5
 6  import java.util.*;
 7
 8  /**
 9   * The Juno shell command to log out.
10   * Usage:
11   * <pre>
12   *      logout
13   * </pre>
14   *
15   * @version 7
16   */
17
18  public class LogoutCommand extends ShellCommand
19  {
20      LogoutCommand()
21      {
22          super( "log out, return to login: prompt" );
23      }
24
25      /**
26       * Log out from the current shell.
27       *
28       * @param args the remainder of the command line.
29       * @param sh   the current shell
30       *
31       * @exception JunoException for reporting errors
32       */
33
34      public void doIt( StringTokenizer args, Shell sh )
35           throws JunoException
36      {
37          throw new ExitShellException();
38      }
39  }
```

```
 1  // joi/7/juno/RemoveCommand.java
 2  //
 3  //
 4  // Copyright 2003, Bill Campbell and Ethan Bolker
 5
 6  import java.util.*;
 7
 8  /**
 9   * The Juno shell command to remove a text file.
10   * Usage:
11   * <pre>
12   *     remove textfile
13   * </pre>
14   *
15   * @version 7
16   */
17
18  public class RemoveCommand extends ShellCommand
19  {
20      RemoveCommand()
21      {
22          super( "remove a TextFile", "textfile" );
23      }
24
25      /**
26       * Remove a TextFile.
27       *
28       * @param args the remainder of the command line.
29       * @param sh   the current Shell
30       *
31       * @exception JunoException for reporting errors
32       */
33
34      public void doIt( StringTokenizer args, Shell sh )
35           throws JunoException
36      {
37          String filename = args.nextToken();
38          sh.getDot().removeJFile(filename);
39      }
40  }
```

```
// joi/7/jfiles/JFile.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.Date;
import java.io.File;

/**
 * A JFile object models a file in a hierarchical file system.
 * <p>
 * Extend this abstract class to create particular kinds of JFiles,
 * e.g.:<br>
 * Directory  -
 *    a JFile that maintains a list of the files it contains.<br>
 * TextFile  -
 *    a JFile containing text you might want to read.<br>
 *
 * @see Directory
 * @see TextFile
 *
 * @version 7
 */

public abstract class JFile
{
    /**
     * The separator used in pathnames.
     */

    public static final String separator = File.separator;

    private String    name;       // a JFile knows its name
    private User      owner;      // the owner of this file
    private Date      createDate; // when this file was created
    private Date      modDate;    // when this file was last modified
    private Directory parent;     // the Directory containing this file

    /**
     * Construct a new JFile, set owner, parent, creation and
     * modification dates. Add this to parent (unless this is the
     * root Directory).
     *
     * @param name     the name for this file (in its parent directory).
     * @param creator  the owner of this new file.
     * @param parent   the Directory in which this file lives.
     */

    protected JFile( String name, User creator, Directory parent )
    {
        this.name   = name;
        this.owner  = creator;
        this.parent = parent;
        if (parent != null) {
            parent.addJFile( name, this );
        }
```



```
        createDate  = modDate = new Date(); // set dates to now
    }

    /**
     * The name of the file.
     *
     * @return the file's name.
     */

    public String getName()
    {
        return name;
    }

    /**
     * The full path to this file.
     *
     * @return the path name.
     */

    public String getPathName()
    {
        if (this.isRoot()) {
            return separator;
        }
        if (parent.isRoot()) {
            return separator + getName();
        }
        return parent.getPathName() + separator + getName();
    }

    /**
     * The size of the JFile
     * (as defined by the child class)..
     *
     * @return the size.
     */

    public abstract int getSize();

    /**
     * Suffix used for printing file names
     * (as defined by the child class).
     *
     * @return the file's suffix.
     */

    public abstract String getSuffix();

    /**
     * Set the owner for this file.
     *
     * @param owner the new owner.
     */

    public void setOwner( User owner )
```

```
113      {
114          this.owner = owner;
115      }
116
117      /**
118       * The file's owner.
119       *
120       * @return the owner of the file.
121       */
122
123      public User getOwner()
124      {
125          return owner;
126      }
127
128      /**
129       * The date and time of the file's creation.
130       *
131       * @return the file's creation date and time.
132       */
133
134      public String getCreateDate()
135      {
136          return createDate.toString();
137      }
138
139      /**
140       * Set the modification date to "now".
141       */
142
143      protected void setModDate()
144      {
145          modDate = new Date();
146      }
147
148      /**
149       * The date and time of the file's last modification.
150       *
151       * @return the date and time of the file's last modification.
152       */
153
154      public String getModDate()
155      {
156          return modDate.toString();
157      }
158
159      /**
160       * The Directory containing this file.
161       *
162       * @return the parent directory.
163       */
164
165      public Directory getParent()
166      {
167          return parent;
168      }
```



```
169
170      /**
171       * A JFile whose parent is null is defined to be the root
172       * (of a tree).
173       *
174       * @return true when this JFile is the root.
175       */
176
177      public boolean isRoot()
178      {
179          return (parent == null);
180      }
181
182      /**
183       * How a JFile represents itself as a String.
184       * That is,
185       * <pre>
186       *    owner   size    modDate    name+suffix
187       * </pre>
188       *
189       * @return the String representation.
190       */
191
192      public String toString()
193      {
194          return getOwner() + "\t" +
195              getSize()  + "\t" +
196              getModDate()  + "\t" +
197              getName() + getSuffix();
198      }
199  }
```

```
// joi/7/juno/Directory.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Directory of JFiles.
 *
 * A Directory is a JFile that maintains a
 * table of the JFiles it contains.
 *
 * @version 7
 */

public class Directory extends JFile
{
    private TreeMap jfiles;  // table for JFiles in this Directory

    /**
     * Construct a Directory.
     *
     * @param name the name for this Directory (in its parent Directory)
     * @param creator the owner of this new Directory
     * @param parent the Directory in which this Directory lives.
     */

    public Directory( String name, User creator, Directory parent)
    {
        super( name, creator, parent );
        jfiles = new TreeMap();
    }

    /**
     * The size of a Directory is the number of JFiles it contains.
     *
     * @return the Directory's size.
     */

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

    /**
     * Suffix used for printing Directory names;
     * we define it as the (system dependent)
     * name separator used in path names.
     *
     * @return the suffix for Directory names.
     */

    public String getSuffix()
    {
        return  JFile.separator;
```



```
    }

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

    public void addJFile(String name, JFile afile)
    {
        jfiles.put( name, afile );
        setModDate();
    }

    /**
     * Get a JFile in this Directory, by name .
     *
     * @param filename the name of the JFile to find.
     * @return the JFile found.
     */

    public JFile retrieveJFile( String filename )
    {
        JFile aFile = (JFile)jfiles.get( filename );
        return aFile;
    }

    /**
     * Remove a JFile in this Directory, by name .
     *
     * @param filename the name of the JFile to remove
     */

    public void removeJFile( String filename )
    {
        jfiles.remove( filename );
    }

    /**
     * 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()
    {
        return (String[])jfiles.keySet().toArray( new String[0] );
    }
}
```

```
// joi/7/juno/TextFile.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * A TextFile is a JFile that holds text.
 *
 * @version 7
 */

public class TextFile extends JFile
{
    private String contents;  // The text itself

    /**
     * Construct a TextFile with initial contents.
     *
     * @param name the name for this TextFile (in its parent Directory)
     * @param creator the owner of this new TextFile
     * @param parent  the Directory in which this TextFile lives.
     * @param initialContents the initial text
     */

    public TextFile( String name, User creator, Directory parent,
              String initialContents )
    {
        super( name, creator, parent );
        setContents( initialContents );
    }

    /**
     * Construct an empty TextFile.
     *
     * @param name the name for this TextFile (in its parent Directory)
     * @param creator the owner of this new TextFile
     * @param parent  the Directory in which this TextFile lives
     */

    TextFile( String name, User creator, Directory parent )
    {
        this( name, creator, parent, "" );
    }

    /**
     * The size of a text file is the number of characters stored.
     *
     * @return the file's size.
     */

    public int getSize()
    {
        return contents.length();
    }

    /**
```



```
     * Suffix used for printing text file names is "".
     *
     * @return an empty suffix (for TextFiles).
     */

    public String getSuffix()
    {
        return "";
    }

    /**
     * Replace the contents of the file.
     *
     * @param contents the new contents.
     */

    public void setContents( String contents )
    {
        this.contents = contents;
        setModDate();
    }

    /**
     * The contents of a text file.
     *
     * @return String contents of the file.
     */

    public String getContents()
    {
        return contents;
    }

    /**
     * Append text to the end of the file.
     *
     * @param  text the text to be appended.
     */

    public void append( String text )
    {
        setContents( contents + text );
    }


    /**
     * Append a new line of text to the end of the file.
     *
     * @param  text the text to be appended.
     */

    public void appendLine( String text )
    {
        this.setContents(contents + '\n' + text);
    }
}
```

```
 1  // joi/7/juno/User.java
 2  //
 3  //
 4  // Copyright 2003 Ethan Bolker and Bill Campbell
 5
 6  /**
 7   * Model a Juno user.  Each User has a login name,
 8   * a home directory, and a real name.
 9   *
10   * @version 7
11   */
12
13  public class User
14  {
15      private String name;     // the User's login name
16      private Directory home;  // her home Directory
17      private String realName; // her real name
18
19      /**
20       * Construct a new User.
21       *
22       * @param name     the User's login name.
23       * @param home     her home Directory.
24       * @param realName her real name.
25       */
26
27      public User( String name, Directory home, String realName )
28      {
29          this.name     = name;
30          this.home     = home;
31          this.realName = realName;
32      }
33
34      /**
35       * Get the User's login name.
36       *
37       * @return the name.
38       */
39
40      public String getName()
41      {
42          return name;
43      }
44
45      /**
46       * Convert the User to a String.
47       * The String representation for a User is her
48       * login name.
49       *
50       * @return the User's name.
51       */
52
53      public String toString()
54      {
55          return getName();
56      }
```



```
57
58      /**
59       * Get the User's home Directory.
60       *
61       * @return the home Directory.
62       */
63
64      public Directory getHome()
65      {
66          return home;
67      }
68
69      /**
70       * Get the user's real name.
71       *
72       * @return the real name.
73       */
74
75      public String getRealName()
76      {
77          return realName;
78      }
79  }
```

```
// joi/7/juno/JunoException.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A general Juno Exception.
 *
 * @version 7
 */

public class JunoException extends Exception
{
    /**
     * The default (no argument) constructor.
     */

    public JunoException()
    {
    }

    /**
     * A general Juno exception holding a String message.
     *
     * @param message the message.
     */

    public JunoException( String message )
    {
        // Exception (actually Throwable, Exceptions's superclass)
        // can remember the String passed its constructor.

        super( message );
    }

    // Note, to get the message stored in a JunoException
    // we can just use the (inherited) methods getMessage(),
    // and toString().
}
```

```
 1  // joi/7/juno/BadShellCommandException.java
 2  //
 3  //
 4  // Copyright 2003 Ethan Bolker and Bill Campbell
 5
 6  /**
 7   * The Exception generated when a ShellCommand is misused.
 8   *
 9   * @version 7
10   */
11
12  class BadShellCommandException extends JunoException
13  {
14      ShellCommand command;
15
16      /**
17       * Construct a new BadShellCommandException
18       * containing the badly used command.
19       *
20       * @param the ShellCommand being misused.
21       */
22
23      public BadShellCommandException( ShellCommand command )
24      {
25          this.command = command;
26      }
27
28      /**
29       * Get the command.
30       */
31
32      public ShellCommand getCommand()
33      {
34          return command;
35      }
36  }
```

```
 1  // joi/7/juno/ExitShellException.java
 2  //
 3  //
 4  // Copyright 2003 Bill Campbell and Ethan Bolker
 5
 6  /**
 7   * Exception raised for exiting a shell.
 8   *
 9   * @version 7
10   */
11
12  public class ExitShellException extends JunoException
13  {
14  }
```