/** A Bank object simulates the behavior of a simple bank/ATM. It contains a Terminal object and a collection of BankAccount objects. */

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 done

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

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

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

    public Bank( String bankName, Terminal atm ) {
        this.atm      = atm;
        this.bankName = bankName;
        // initialize collection:
        accountList   = new BankAccount[ NUM_ACCOUNTS ]; ///
        accountList[0] = new BankAccount(  0, this);
        accountList[1] = new BankAccount(100, this);
        accountList[2] = new BankAccount(200, this);
    }

    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 );
        }
    }

    private void openNewAccount() {
        /** Open a new bank account, prompting the user for information. */
    }

    private void report() {
        /** Print detailed report. */
    }

    private BankAccount whichAccount() {
        /** Find the a BankAccount for the customer. */
    }

    private void processTransactionsForAccount( BankAccount acct ) {
        /** Process all transactions for a given account. */
    }

    private void help( String commands ) {
        /** Print the current help information. */
    }

    private void instructUser() {
        /** Print an introductory message. */
    }

    static {
        TERMINAL = new Terminal("Bank", "Banker commands: exit, open, customer, report, help.");
        ACCOUNTS = new BankAccount[NUM_ACCOUNTS];
    }
```java
private void openNewAccount()
{
    // when accountList is a dynamic collection
    // remove the next two lines, uncomment and complete
    // the code between /* and */
    atm.println(bankName + " is accepting no new customers
    return;
    
    /*
    // prompt for initial deposit
    int startup = atm.readInt( "Initial deposit: " );
    
    // create newAccount
    BankAccount newAccount = new BankAccount( startup, this );
    
    //  and add it to accountList
    ???
    
    // inform user
    atm.println( "opened new account " + ??? /// name or number
    + " with $" + newAccount.getBalance());
    */
    
    // Prompt the customer for transaction to process.
    // Then send an appropriate message to the account.
    private void processTransactionsForAccount( BankAccount acct )
    {
        help( CUSTOMER_TRANSACTIONS );
        
        String transaction;
        while (!(transaction =
            atm.readWord("    transaction: ").equals("quit")) ) {
            
            if ( transaction.startsWith( "h" ) ) {
                help( CUSTOMER_TRANSACTIONS );
            }
            else if ( transaction.startsWith( "d" ) ) {
                int amount = atm.readInt( "    amount: " );
                atm.println("    deposited " + acct.deposit( amount ));
            }
            else if ( transaction.startsWith( "w" ) ) {
                int amount = atm.readInt( "    amount: " );
                atm.println("    withdrew " + acct.withdraw( amount ));
            }
            else if (transaction.startsWith("t")) {
                atm.print( "    to ");
                BankAccount toacct = whichAccount();
                if (toacct != null) {
                    int amount = atm.readInt("    amount to transfer: ");
                    atm.println("    transfered " +
                        toacct.deposit(acct.withdraw(amount)));
                }
            }
            else if (transaction.startsWith("b")) {
                atm.println("    current balance " +
                    acct.requestBalance());
            }
            else {
                atm.println("    sorry, unknown transaction" );
            }
        }
        atm.println();
    }
    
    // Prompt for an account name (or number), look it up
    // in the account list. If it's there, return it;
    // otherwise report an error and return null.
    
    private BankAccount whichAccount()
    {
        // prompt for account name or account number
        // (whichever is appropriate)
        int accountNumber = atm.readInt("account number: ");
        
        // look up account in accountList
        // if it's there, return it
        // else the following two lines should execute
        if ( accountNumber >= 0 && accountNumber < NUM_ACCOUNTS ) {
            return accountList[accountNumber];
        }
        else {
            atm.println("not a valid account");
            return null;
        }
    }
    
    // Report bank activity.
    // For each BankAccount, print the customer id (name or number),
    // account balance and the number of transactions.
    // Then print bank totals.
    
    private void report()
    {
        atm.println( "Summaries of individual accounts:" );
        atm.println( "account  balance   transaction count" );
        
        for (int i = 0; i < NUM_ACCOUNTS; i++ ) {
            atm.println(i + "	" + accountList[i].getBalance() +
                "	" + accountList[i].getTransactionCount());
        }
        
        atm.println( "Bank totals:");
        atm.println( "open accounts: " + getNumberOfAccounts() );
        atm.println( "cash on hand: $" + getBalance());
        atm.println( "transactions:  " + getTransactionCount());
        atm.println();
    }

    // Welcome the user to the bank and instruct her on
    }
```
private void instructUser()
{
atm.println( "Welcome to " + bankName);
atm.println( "Open some accounts and work with them." );
help( BANKER_COMMANDS );
}

private void help( String helpString )
{
atm.println( helpString );
atm.println();
}

/**
* Increment bank balance by given amount.
*
* @param amount the amount increment.
*/

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

/**
* Increment by one the count of transactions,
* for this bank.
*/

public void countTransaction()
{
transactionCount++;
}

/**
* Get the number of transactions performed by this bank.
*
* @return number of transactions performed.
*/

public int getTransactionCount( )
{
return transactionCount ;
}

/**
* Get the current bank balance.
*
* @return current bank balance.
*/

public int getBalance()
{
return balance;
}

/**
* Get the current number of open accounts.
*
* @return number of open accounts.
*/

public int getNumberOfAccounts()
{
return NUM_ACCOUNTS; /// needs changing ...
}

public static void main( String[] args )
{
// parse the command line arguments for the echo
// flag and the name of the bank

boolean echo    = false; // default does not echo
String bankName = "River Bank"; // default bank name

for (int i = 0; i < args.length; i++ ) {
if (args[i].equals("-e")) {
echo = true;
} else {
bankName = args[i];
}
}
Bank aBank = new Bank( bankName, new Terminal(echo) );
aBank.visit();
}
/*
 * 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
 * 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 performed.
    private Bank issuingBank;          // Bank issuing this account

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

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

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

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

    public int getBalance() {
        return balance;
    }

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

    public int getTransactionCount() {
        return transactionCount;
    }

    public Bank getIssuingBank() {
        return issuingBank;
    }

    public void setIssuingBank(Bank issuingBank) {
        this.issuingBank = issuingBank;
    }
}

/*
 * end of BankAccount.java Page 2
 */

public int getTransactionCount()
{
    return transactionCount;
}

/**
 * Increment by 1 the count of transactions, for this account
 * and for the issuing Bank.
 * Does NOT count as a transaction.
 */

public void countTransaction()
{
    transactionCount++;
    this.getIssuingBank().countTransaction();
}

/**
 * Get the bank that issued this account.
 * Does NOT count as a transaction.
 *
 * @return issuing bank.
 */

public Bank getIssuingBank()
{
    return issuingBank;
}
1 open
2 1000
3 open
4 2000
5 help
6 report
7 balance
customer balance deposit quit customer transfer transfer 45 quit 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:

<table>
<thead>
<tr>
<th>account</th>
<th>balance</th>
<th>transaction count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$1000</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>$2000</td>
<td>1</td>
</tr>
</tbody>
</table>

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, help.

transaction: balance
current balance 1000

transaction: deposit
amount: 9999
deposited 9999

current balance 10999

transaction: quit

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

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

Goodbye from River Bank
1. open
groucho
1000
customer
harpo
open
harpo
2000
help
report
open
chico
3000
customer
groucho
balance
deposit
9999
balance
deposit
9999
groucho
customer
chico
transfer
45
quit
customer
harpo
transfer
45
quit
customer
groucho
transfer
45
quit
customer
harp
open
chico
customer
harp
open
chico
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:

<table>
<thead>
<tr>
<th>account</th>
<th>balance</th>
<th>transaction count</th>
</tr>
</thead>
<tbody>
<tr>
<td>groucho</td>
<td>$1000</td>
<td>1</td>
</tr>
<tr>
<td>harpo</td>
<td>$2000</td>
<td>1</td>
</tr>
</tbody>
</table>

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

does not work with $5000

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, help.

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, help.

transaction: transfer
to account name: chico
amount to transfer: 45
transferred 45

transaction: quit

Summaries of individual accounts:

<table>
<thead>
<tr>
<th>account</th>
<th>balance</th>
<th>transaction count</th>
</tr>
</thead>
<tbody>
<tr>
<td>chico</td>
<td>$3045</td>
<td>2</td>
</tr>
<tr>
<td>groucho</td>
<td>$10999</td>
<td>4</td>
</tr>
<tr>
<td>harpo</td>
<td>$1955</td>
<td>2</td>
</tr>
</tbody>
</table>

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

Goodbye from River Bank
import java.util.ArrayList;

/**
* Reverse the order of lines entered from standard input.
*/
public class Reverse {
    public static void main( String[] args ) {
        Terminal t    = new Terminal();
        ArrayList list = new ArrayList();
        String line;

        while ((line = t.readLine()) != null ) {
            list.add(line);
        }

        for (int i = list.size()-1; i >= 0; i--) {
            line = (String)list.get(i);
            t.println( line );
        }
    }
}
/* return the number of words */
/* get the size of the Dictionary */
/**
 * return (definition.equals(Definition.get(look up any entry in the Dictionary).get(word)) ? null : word)
 */
 /* return definition of the word, null if none */
 /* throw definition of the word whose definition is sought */
 /* add an entry to this Dictionary */
 **/ 

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

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

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

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

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 + ":
" + definition.toString() + "\n";
}
return str;
}
package dictionary;

/**
 * Model the definition of a word in a dictionary.
 *
 * @see Dictionary
 *
 * @version 4
 */

public class Definition
{
private String definition;    // the defining string

/**
 * Construct a simple Definition.
 *
 * @param definition the definition.
 */

public Definition( String definition )
{
this.definition = definition;
}

/**
 * Construct a String representation of this Definition.
 *
 * @return the definition string.
 */

public String toString()
{
return definition;
}
}
public class Lookup
{
  private static Terminal t = new Terminal();
  private static Dictionary dictionary = new Dictionary();

  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" ) );
  }

  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());
    }
  }

  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(dictionary.toString());
        t.println("-------------------");
      }
      else {
        t.println(word + ":");
        printDefinition(word);
      }
    }

    // words entered interactively
  }
}
```java
public class Lookup {

    public static void main(String[] args) {
        System.out.println("Look up words, "quit" to quit");
        while (true) {
            String word = scanner.nextLine(); // Read word from user
            if (word.equals("quit")) { // Break if user enters "quit"
                break;
            }
            printDefinition(word); // Print definition of word
        }
    }

    public static void printDefinition(String word) {
        // Code to print definition of a word
    }
}
```
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.Date;

/**
 * A TextFile mimics the sort of text file that one finds
 * on a computer's file system.  It has an owner,
 * a create date (when the file was created),
 * a modification date (when the file was last modified),
 * and String contents.
 *
 * @version 3
 */

public class TextFile {

    // Private Implementation

    private String owner;       // Who owns the file.
    private Date   createDate;  // When the file was created.
    private Date   modDate;     // When the file was last modified.
    private String contents;    // The text stored in the file.

    // Public Interface

    /**
     * Construct a new TextFile with given owner and
     * contents; set the creation and modification dates.
     *
     * @param owner the user who owns the file.
     * @param contents the file's initial contents.
     */

    public TextFile( String owner, String contents ) {
        this.owner    = owner;
        this.contents = contents;
        createDate    = new Date(); // date and time now
        modDate       = createDate;
    }

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

    public void setContents( String contents ) {
        this.contents = contents;
        modDate = new Date();
    }

    /**
     * The contents of a 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 ) {
        this.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 + '
' + text);
    }

    /**
     * The size of a file.
     *
     * @return the integer size of the file
     *  (the number of characters in its String contents)
     */

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

    /**
     * The data and time of the file's creation.
     *
     * @return the file's creation date and time.
     */

    public String getCreateDate() {
        return createDate.toString();
    }
}

public class TextFileTest {  // A test class

    public static void main(String[] args) {
        TextFile tf = new TextFile( "Owner", "Some Contents" );
        System.out.println( tf.getContents() );
        tf.setContents( "New Contents" );
        System.out.println( tf.getContents() );
        tf.append( "More Text" );
        System.out.println( tf.getContents() );
        tf.appendLine( "Line 1" );
        System.out.println( tf.getContents() );
        System.out.println( tf.getSize() );
        System.out.println( tf.getCreateDate() );
    }
}
public String getModDate()
{
    return modDate.toString();
}

public String getOwner()
{
    return owner;
}

public static void main( String[] args )
{
    Terminal terminal = new Terminal();
    TextFile myTextFile = new TextFile( "bill", "Hello, world." );

    terminal.println( "TextFile myTextFile contains " + myTextFile.getSize() + " characters." );
    terminal.println( "Created by " + myTextFile.getOwner() + ", " + myTextFile.getCreateDate() );
    terminal.println( myTextFile.getContents() );
    terminal.println();

    terminal.println( "append new line " + "How are you today?" );
    myTextFile.appendLine( "How are you today?" );
    terminal.println( myTextFile.getContents() );
    terminal.println( "TextFile myTextFile contains " + myTextFile.getSize() + " characters." );
    terminal.println( "Modified " + myTextFile.getModDate() );
}
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * 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>
 */

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

}
Listing 4.13 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;

    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 ) );
    }

    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
            }
            else if (whoAreYou.equals("manager")) {
                managerVisit();
            }
            else 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() {
        ShoppingCart basket = new ShoppingCart();
        browser.println( "Currently available:");
        catalog.show(browser);
        while ( true ) { // loop forever ...
            String nextPurchase = browser.readWord( "Next purchase: ");
            if (nextPurchase.equals("break")) {
                break; // leave the while loop
            }
            else if (nextPurchase.equals("manage")) {
                if (managerVisit()) {
                    continue; // go back to the customer's shoppingCart
                }
            } else if (nextPurchase.equals("visit")) {
                customerVisit();
            }
            else { // add an Item to the catalog
                Item newItem = new Item( browser.readLine(" name: "), browser.readInt(" cost: ") );
                if (catalog.add( newItem )) {
                    basket.addItem( newItem );
                    browser.println("Item added to shoppingCart.");
                } else {
                    browser.println("Item already in catalog.");
                }
            }
        }
    }
}

// End EStore.java

/** End copyright notice */
```java
if ( nextPurchase.equals("checkout")) break; // leave loop!

if ( nextPurchase.equals("help")) {
    catalog.show(browser);
    continue; // go back to top of while loop
}

// customer has entered the name of an Item
basket.addItem( catalog.getItem(nextPurchase) );

int numberPurchased = basket.getCount();

browser.println("We are shipping these ", numberPurchased, " Items:");
basket.showContents(browser);
browser.println("and charging your account ", basket.getCost());

browser.println("Thank you for shopping at ", storeName);
}

/**
 *  The EStore simulation program begins here when the user
 *  issues the command <code>java EStore</code>
 *
 *  If first command line argument is 
 *  then look at second for store name
 *
 *  The next command line argument is the name of the EStore.
 *
 *  @param args <-e> <storeName>
 */
public static void main( String[] args )
{
    String storeName = "Virtual Minimal Mall"; //default

    // check to see if first argument was 
    boolean echo = ( (args.length > 0) && (args[0].equals("-e")) );

    // if first argument was 
    int nextArg = (echo ? 1 : 0 );

    if (args.length > nextArg) {
        storeName = args[nextArg];
    }

    // Print this to simulate internet search.
    System.out.println("connecting ... ");

    // Create an EStore object and visit it
    (new EStore(storeName, new Terminal(echo))).visit();
    }
}
// joistore/ShoppingCart.java

/**
 * A ShoppingCart keeps track of a customer's purchases.
 * @see EStore
 * @version 4
 */

public class ShoppingCart {
    private int count; // number of Items in this ShoppingCart
    private int cost;  // total cost of Items in this ShoppingCart

    public ShoppingCart() {
        count = 0;
        cost  = 0;
    }

    public void addItem( Item item ) {
        count++;
        this.cost += item.getCost(); // Java idiom: a += b means a = a + b
    }

    public void returnItem( Item item ) {
        /// look through the list looking for Item
        /// remove it if it's there
    }

    public int getCount() {
        /// get this information from the list,
        /// since the count field no longer exists
        return count;
    }

    public int getCost() {
        /// get this information from the list,
        /// since the cost field no longer exists
        return cost;
    }

    public void showContents( Terminal t) {
        /// work to do here ...
        t.println("[sorry, can't yet print ShoppingCart contents]");
    }
}

*/
public class Item {
    private int cost;
    private String name;

    public Item(String name, int cost) {
        this.name = name;
        this.cost = cost;
    }

    public int getCost() {
        return cost;
    }

    public String getName() {
        return name;
    }
}
import java.util.TreeMap;

/**
 * A Catalog models the collection of Items that an
 * EStore might carry.
 *
 * @see EStore
 *
 * @version 4
 */

public class Catalog
{
private TreeMap items;

/**
 * Construct a Catalog object.
 */

public Catalog()
{
items = new TreeMap();
}

/**
 * Add an Item to this Catalog.
 *
 * @param item the Item to add.
 */

public void addItem(Item item)
{
items.put(item.getName(), item);
}

/**
 * Get an Item from this Catalog.
 *
 * @param itemName the name of the wanted Item
 *
 * @return the Item, null if none.
 */

public Item getItem(String itemName)
{
return (Item)items.get(itemName);
}

/**
 * Display the contents of this Catalog.
 *
 * @param t the Terminal to print to.
 */

public void show(Terminal t)
{
// loop on items, printing name and cost

// [sorry, can't yet print Catalog contents]
}

}