```java
// joi/9/copy/Copy1.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.io.*;

/**
 * Simple read-a-char, write-a-char loop to exercise file I/O.
 *
 * Usage: java Copy1 inputfile outputfile
 */

public class Copy1
{
    private static final int EOF = -1;  // end of file character rep.

    /**
     * All work is done here.
     *
     * @param args names of the input file and output file.
     */

    public static void main( String[] args )
    {
        FileReader inStream  = null;
        FileWriter outStream = null;
        int ch;

        try {
            // open the files
            inStream  = new FileReader( args[0] );
            outStream = new FileWriter( args[1] );

            // copy
            while ((ch = inStream.read()) != EOF) {
                outStream.write( ch );
            }
        }
        catch (IndexOutOfBoundsException e) {
            System.err.println(
                "usage: java Copy1 sourcefile targetfile" );
        }
        catch (FileNotFoundException e) {
            System.err.println( e );  // rely on e’s toString()
        }
        catch (IOException e) {
            System.err.println( e );
        }
        finally { // close the files
            try {
                if (inStream != null) {
                    inStream.close();
                }
            }
            catch (Exception e) {
```

```java
                System.err.println("Unable to close input stream.");
            }
            try {
                if (outStream != null) {
                    outStream.close();
                }
            }
            catch (Exception e) {
                System.err.println("Unable to close output stream.");
            }
        }
    }
}
```

```java
// joi/9/copy/Copy2.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.io.*;

/**
 * Simple read-a-line write-a-line loop to exercise file I/O.
 *
 * Usage: java Copy2 inputfile outputfile
 */

public class Copy2
{
    /**
     * All work is done here.
     *
     * @param args names of the input file and output file.
     */

    public static void main( String[] args )
    {
        BufferedReader inStream  = null;
        BufferedWriter outStream = null;
        String line;

        try {
            // open the files
            inStream =  new BufferedReader(new FileReader(args[0]));
            outStream = new BufferedWriter(new FileWriter(args[1]));

            // copy
            while ((line = inStream.readLine()) != null) {
                outStream.write( line );
                outStream.newLine();
            }
        }
        catch (IndexOutOfBoundsException e) {
            System.err.println(
                "usage: java Copy2 sourcefile targetfile" );
        }
        catch (FileNotFoundException e) {
            System.err.println( e );  // rely on e’s toString()
        }
        catch (IOException e) {
            System.err.println( e );
        }
        finally { // close the files
            try {
                if (inStream != null) {
                    inStream.close();
                }
            }
            catch (Exception e) {
                System.err.println("Unable to close input stream.");
```

```java
            }
            try {
                if (outStream != null) {
                    outStream.close();
                }
            }
            catch (Exception e) {
                System.err.println("Unable to close output stream.");
            }
        }
    }
}
```

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

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

/**
 * A Bank object simulates the behavior of a simple bank/ATM.
 * It contains a Terminal object and a collection of
 * BankAccount objects.
 *
 * The visit method opens this Bank for business,
 * prompting the customer for input.
 *
 * It is persistent: it can save its state to a file and read it
 * back at a later time.
 *
 * To create a Bank and open it for business issue the command
 * <code>java Bank</code> with appropriate arguments.
 *
 * @see BankAccount
 * @version 9
 */

public class Bank
    implements Serializable
{
    private String bankName;           // the name of this Bank
    private transient Terminal atm;   // for communication with world
    private int balance = 0;           // total cash on hand
    private int transactionCount = 0;  // number of Bank transactions
    private Month month;               // the current month.
    private Map accountList;           // mapping names to accounts.

    private int checkFee = 2;            // cost for each check
    private int transactionFee = 1;      // fee for each transaction
    private int monthlyCharge = 5;       // monthly charge
    private double interestRate = 0.05;  // annual rate paid on savings
    private int maxFreeTransactions = 3; // for savings accounts

    // what the banker can ask of the bank

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

    // what the customer can ask of the bank

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

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

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

    public Bank( String bankName )
    {
        this.atm      = atm;
        this.bankName = bankName;
        accountList   = new TreeMap();
        month         = new Month();
    }

    /**
     * Simulates interaction with a Bank.
     * Presents the user with an interactive loop, prompting for
     * banker transactions and in the case of the banker
     * transaction "customer", an account id and further
     * customer transactions.
     */

    public void visit()
    {
        instructUser();

        String command;
        while (!(command =
                 atm.readWord("banker command: ")).equals("exit")) {

            if (command.startsWith("h")) {
                help( BANKER_COMMANDS );
            }
            else if (command.startsWith("o")) {
                openNewAccount();
            }
            else if (command.startsWith("n")) {
                newMonth();
            }
            else if (command.startsWith("r")) {
                report();
            }
            else if (command.startsWith( "c" ) ) {
                BankAccount acct = whichAccount();
                if ( acct != null ) {
                        processTransactionsForAccount( acct );
                }
            }
            else {
                // Unrecognized Request
                atm.println( "unknown command: " + command );
            }
        }
        report();
        atm.println( "Goodbye from " + bankName );
    }


    // Open a new bank account,
    // prompting the user for information.

    private void openNewAccount()
    {
        String accountName = atm.readWord( "Account name: " );
        char accountType =
            atm.readChar( "Type of account (r/c/f/s): " );
        try {
            int startup = readPosAmt( "Initial deposit: " );
            BankAccount newAccount;
            switch( accountType ) {
            case ’c’:
                newAccount = new CheckingAccount( startup, this );
                break;
            case ’f’:
                newAccount = new FeeAccount( startup, this );
                break;
            case ’s’:
                newAccount = new SavingsAccount( startup, this );
                break;
            case ’r’:
                newAccount = new RegularAccount( startup, this );
                break;
            default:
                atm.println("invalid account type: " + accountType);
                return;
            }
            accountList.put( accountName, newAccount );
            atm.println( "opened new account " + accountName
                         + " with $" + startup );
        }
        catch (NegativeAmountException e) {
            atm.errPrintln(
              "You cannot open an account with a negative balance");
        }
        catch (InsufficientFundsException e) {
            atm.errPrintln( "Initial deposit doesn’t cover fee" );
        }
    }

    // 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")) {

            try {
                if ( transaction.startsWith( "h" ) ) {
                    help( CUSTOMER_TRANSACTIONS );
                }
```

```java
                else if ( transaction.startsWith( "d" ) ) {
                    int amount = readPosAmt( "    amount: " );
                    atm.println("    deposited "
                                + acct.deposit( amount ));
                }
                else if ( transaction.startsWith( "w" ) ) {
                    int amount = readPosAmt( "    amount: " );
                    atm.println("    withdrew "
                                + acct.withdraw( amount ));
                }
                else if ( transaction.startsWith( "c" ) ) {
                    int amount = readPosAmt( "    amount of check: " );
                    try { // to cast acct to CheckingAccount ...
                        atm.println("    cashed check for " +
                           ((CheckingAccount) acct).honorCheck( amount ))
                    }
                    catch (ClassCastException e) {
                        // if not a checking account, report error
                        atm.errPrintln(
                            "    Sorry, not a checking account." );
                    }
                }
                else if (transaction.startsWith("t")) {
                    atm.print( "    to ");
                    BankAccount toacct = whichAccount();
                    if (toacct != null) {
                    int amount = readPosAmt("    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" );
                }
            }
            catch (InsufficientFundsException e) {
                atm.errPrintln( "    Insufficient funds " +
                                e.getMessage() );
            }
            catch (NegativeAmountException e) {
              atm.errPrintln("    Sorry, negative amounts disallowed." );
            }
            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()
    {
        String accountName = atm.readWord( "account name: " );
        BankAccount account = (BankAccount) accountList.get(accountName);
        if (account == null) {
            atm.println( "not a valid account" );
        }
        return account;
    }

    // Action to take when a new month starts.
    // Update the month field by sending a next message.
    // Loop on all accounts, sending each a newMonth message.

    private void newMonth()
    {
        month.next();
        Iterator i = accountList.keySet().iterator();
        while ( i.hasNext() ) {
            String name = (String) i.next();
            BankAccount acct = (BankAccount) accountList.get( name );
            try {
                acct.newMonth();
            }
            catch (InsufficientFundsException exception) {
                atm.errPrintln( "Insufficient funds in account \"" +
                                name + "\" for monthly fee" );
            }
        }
    }

    // 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( bankName + " report for " + month );
        atm.println( "\nSummaries of individual accounts:" );
        atm.println( "account  balance   transaction count" );
        for (Iterator i = accountList.keySet().iterator();
             i.hasNext(); ) {
            String accountName = (String) i.next();
            BankAccount acct = (BankAccount) accountList.get(accountName)
            atm.println(accountName + "\t$" + acct.getBalance() + "\t\t"
                        + acct.getTransactionCount());
        }
        atm.println( "\nBank 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
    //  her options.
```

```java

    private void instructUser()
    {
        atm.println( "Welcome to " + bankName );
        atm.println( month.toString() );
        atm.println( "Open some accounts and work with them." );
        help( BANKER_COMMANDS );
    }

    // Display a help string.

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

    // Read amount prompted for from the atm.
    // Throw a NegativeAmountException if amount < 0

    private int readPosAmt( String prompt )
         throws NegativeAmountException
    {
        int amount = atm.readInt( prompt );
        if (amount < 0) {
            throw new NegativeAmountException();
        }
        return amount;
    }

    /**
     * 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 ;
    }

    /**
     * The charge this bank levies for cashing a check.
     *
     * @return check fee
     */

    public int getCheckFee( )
    {
        return checkFee ;
    }

    /**
     * The charge this bank levies for a transaction.
     *
     * @return the transaction fee
     */

    public int getTransactionFee( )
    {
        return transactionFee ;
    }

    /**
     * The charge this bank levies each month.
     *
     * @return the monthly charge
     */

    public int getMonthlyCharge( )
    {
        return monthlyCharge;
    }

    /**
     * The current interest rate on savings.
     *
     * @return the interest rate
     */

    public double getInterestRate( )
    {
        return interestRate;
    }

    /**
     * The number of free transactions per month.
     *
     * @return the number of transactions
     */

```

```java
    public int getMaxFreeTransactions()
    {
        return maxFreeTransactions;
    }

    /**
     * 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 accountList.size();
    }

    /**
     * Set the atm for this Bank.
     *
     * @param atm the Bank's atm.
     */

    public void setAtm( Terminal atm ) {
        this.atm = atm;
    }


    /**
     * Run the simulation by creating and then visiting a new Bank.
     * <p>
     * A -e argument causes the input to be echoed.
     * This can be useful for executing the program against
     * a test script, e.g.,
     * <pre>
     *   java Bank -e < Bank.in
     * </pre>
     *
     * The -f argument reads the bank's state from the specified
     * file, and writes it to that file when the program exits.
     *
     * @param args the command line arguments:
     *         <pre>
     *         -e echo input.
     *         -f filename
     *         bankName any other command line argument.
     *         </pre>
     */

    public static void main( String[] args )
    {
        boolean echo        = false;
        String bankFileName = null;
        String bankName     = "Persistent Bank";
        Bank theBank        = null;

        // parse the command line arguments
        for (int i = 0; i < args.length; i++ ) {
            if (args[i].equals("-e")) { // echo input to output
                echo = true;
                continue;
            }
            if (args[i].equals("-f")) { // read/write Bank from/to file
                bankFileName = args[++i];
                continue;
            }
        }

        // create a new Bank or read one from a file
        if (bankFileName == null) {
            theBank = new Bank( bankName );
        }
        else {
            theBank = readBank( bankName, bankFileName );
        }

        // give the Bank a Terminal, then visit
        theBank.setAtm(new Terminal(echo));
        theBank.visit();

        // write theBank’s state to a file if required
        if (bankFileName != null) {
            writeBank(theBank, bankFileName);
        }
    }

    // Read a Bank from a file (create it if file doesn’t exist).
    //
    // @param bankName     the name of the Bank
    // @param bankFileName the name of the file containing the Bank
    //
    // @return the Bank

    private static Bank readBank(String bankName, String bankFileName)
    {
        File file = new File( bankFileName );
        if (!file.exists()) {
            return new Bank( bankName );
        }
        ObjectInputStream inStream = null;
        try {
            inStream = new ObjectInputStream(
```

```java
                               new FileInputStream( file ) );
            Bank bank = (Bank)inStream.readObject();
            System.out.println(
               "Bank state read from file " + bankFileName);
            return bank;
        }
        catch (Exception e ) {
            System.err.println(
                "Problem reading " + bankFileName );
            System.err.println(e);
            System.exit(1);
        }
        finally {
            try {
                inStream.close();
            }
            catch (Exception e) {
            }
        }
        return null; // you can never get here
    }


    // Write a Bank to a file.
    //
    // @param bank     the Bank
    // @param fileName the name of the file to write the Bank to

    private static void writeBank( Bank bank, String fileName)
    {
        ObjectOutputStream outStream = null;
        try {
            outStream = new ObjectOutputStream(
                               new FileOutputStream( fileName ) );
            outStream.writeObject( bank );
            System.out.println(
               "Bank state written to file " + fileName);
        }
        catch (Exception e ) {
            System.err.println(
               "Problem writing " + fileName );
        }
        finally {
            try {
                outStream.close();
            }
            catch (Exception e ) {
            }
        }
    }
}
```

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

import java.io.Serializable;

/**
 * A BankAccount object has private fields to keep track
 * of its current balance, the number of transactions
 * performed and the Bank in which it is an account, and
 * and public methods to access those fields appropriately.
 *
 * @see Bank
 * @version 9
 */

public abstract class BankAccount
    implements Serializable
{
    private int  balance = 0;          // Account balance (whole dollars)
    private int  transactionCount = 0; // Number of transactions performe
    private Bank issuingBank;          // Bank issuing this account

    /**
     * Construct a BankAccount with the given initial balance and
     * issuing Bank. Construction counts as this BankAccount’s
     * first transaction.
     *
     * @param initialBalance the opening balance.
     * @param issuingBank the bank that issued this account.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    protected BankAccount( int initialBalance, Bank issuingBank )
        throws InsufficientFundsException
    {
        this.issuingBank = issuingBank;
        deposit( initialBalance );
    }

    /**
     * Get transaction fee.  By default, 0.
     * Override this for accounts having transaction fees.
     *
     * @return the fee.
     */

    protected int getTransactionFee()
    {
        return 0;
    }

    /**
     * The bank that issued this account.
```

```java
     *
     * @return the Bank.
     */

    protected Bank getIssuingBank()
    {
        return issuingBank;
    }

    /**
     * Withdraw the given amount, decreasing this BankAccount’s
     * balance and the issuing Bank’s balance.
     * Counts as a transaction.
     *
     * @param amount the amount to be withdrawn
     * @return amount withdrawn
     *
     * @exception InsufficientFundsException when appropriate.
     */

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

    /**
     * Deposit the given amount, increasing this BankAccount’s
     * balance and the issuing Bank’s balance.
     * Counts as a transaction.
     *
     * @param amount the amount to be deposited
     * @return amount deposited
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public int deposit(int amount)
        throws InsufficientFundsException
    {
        incrementBalance( amount - getTransactionFee() );
        countTransaction();
        return amount ;
    }

    /**
     * Request for balance. Counts as a transaction.
     *
     * @return current account balance.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public int requestBalance()
```

```java
        throws InsufficientFundsException
    {
        incrementBalance( - getTransactionFee() );
        countTransaction();
        return getBalance() ;
    }

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

    public int getBalance()
    {
        return balance;
    }

    /**
     * Increment account balance by given amount.
     * Also increment issuing Bank’s balance.
     * Does NOT count as a transaction.
     *
     * @param amount the amount of the increment.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public final void incrementBalance( int amount )
        throws InsufficientFundsException
    {
        int newBalance = balance + amount;
        if (newBalance < 0) {
            throw new InsufficientFundsException(
                          "for this transaction" );
        }
        balance = newBalance;
        getIssuingBank().incrementBalance( amount );
    }

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

    public int getTransactionCount()
    {
        return transactionCount;
    }

    /**
     * Increment by 1 the count of transactions, for this account
     * and for the issuing Bank.
```

```java
     * Does NOT count as a transaction.
     *
     * @exception InsufficientFundsException when appropriate.
     */

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

    /**
     * Action to take when a new month starts.
     *
     * @exception InsufficientFundsException thrown when funds
     *            on hand are not enough to cover the fees.
     */

    public abstract void newMonth()
        throws InsufficientFundsException;
}
```

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

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

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

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

    private int month;
    private int year;

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

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

    /**
     *  Advance to next month.
     */

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

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

```

```java
    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 );
    }
}
```