```java
// joi/5/shapes/Line.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A Line has a length and a paintChar used to paint
 * itself on a Screen.
 *
 * Subclasses of this abstract class specify the direction
 * of the Line.
 *
 * @version 5
 */

public abstract class Line
{
    protected int  length;      // length in (character) pixels.
    protected char paintChar;   // character used for painting.

    /**
     * Construct a Line.
     *
     * @param length length in (character) pixels.
     * @param paintChar character used for painting this Line.
     */

    protected Line( int length, char paintChar )
    {
        this.length    = length;
        this.paintChar = paintChar;
    }

    /**
     * Get the length of this line.
     *
     * @return the length in (character) pixels.
     */

    public int getLength()
    {
        return length;
    }

    /**
     * Set the length of this line.
     *
     * @param length the new length in (character) pixels.
     */

    public void setLength( int length )
    {
        this.length = length;
    }

    /**
```

```java
     * Get the paintChar of this Line.
     *
     * @return the paintChar.
     */

    public char getPaintChar()
    {
        return paintChar;
    }

    /**
     * Set the paintChar of this Line.
     *
     * @param paintChar the new paintChar.
     */

    public void setPaintChar( char paintChar )
    {
        this.paintChar = paintChar;
    }

    /**
     * Paint this Line on Screen s at position (x,y).
     *
     * @param s the Screen on which this Line is to be painted.
     * @param x the x position for the line.
     * @param y the y position for the line.
     */

    public abstract void paintOn( Screen s, int x, int y );

    /**
     * Paint this Line on Screen s at position (0,0).
     *
     * @param s the Screen on which this Line is to be painted.
     */

    public void paintOn( Screen s )
    {
        paintOn( s, 0, 0 );
    }
}
```

```java
// joi/5/shapes/HLine.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker


/**
 * An HLine is a horizontal Line.
 */

public class HLine extends Line
{
    /**
     * Construct an HLine having a paintChar and a length.
     *
     * @param length length in (character) pixels.
     * @param paintChar character used for painting this Line.
     */

    public HLine( int length, char paintChar )
    {
        super( length, paintChar );
    }

    /**
     * Paint this Line on Screen s at position (x,y).
     *
     * @param screen the Screen on which this Line is to be painted.
     * @param x      the x position for the line.
     * @param y      the y position for the line.
     */

    public void paintOn( Screen screen, int x, int y )
    {
        for ( int i = 0; i < length; i++ )
            screen.paintAt( paintChar, x+i, y );
    }

    /**
     * Unit test for class HLine.
     */

    public static void main( String[] args )
    {
        Terminal terminal = new Terminal();

        terminal.println( "Self documenting unit test of HLine.");
        terminal.println( "The two Screens that follow should match.");
        terminal.println();
        terminal.println( "Hard coded picture:");
        terminal.println( "++++++++++++++++++++++");
        terminal.println( "+xxxxxxxxxx          +");
        terminal.println( "+xxxxx               +");
        terminal.println( "+                    +");
        terminal.println( "+   *****            +");
        terminal.println( "+    1               +");
```

```java
        terminal.println( "+                    +");
        terminal.println( "++++++++++++++++++++++");
        terminal.println();

        terminal.println( "Picture drawn using HLine methods:");
        Screen screen = new Screen( 20, 6 );

        Line hline = new HLine( 10, 'x' );
        hline.paintOn( screen );

        hline.setLength(5);
        hline.paintOn( screen, 0, 1 );

        hline.setPaintChar('*');
        hline.paintOn( screen, 3, 3 );

        hline.setLength(1);
        hline.setPaintChar('1');
        hline.paintOn( screen, 4, 4 );

        screen.draw( terminal );
    }

}
```

```java
// joi/5/shapes/VLine.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A VLine is a vertical Line.
 */

public class VLine extends Line
{
    /**
     * Construct a VLine having a paintChar and a length.
     *
     * @param length length in (character) pixels.
     * @param paintChar character used for painting this Line.
     */

    public VLine( int length, char paintChar )
    {
        super( length, paintChar );
    }

    /**
     * Paint this Line on Screen s at position (x,y).
     *
     * @param screen the Screen on which this Line is to be painted.
     * @param x      the x position for the line.
     * @param y      the y position for the line.
     */

    public void paintOn( Screen screen, int x, int y )
    {
        for ( int i = 0; i < length; i++ )
            screen.paintAt( paintChar, x, y+i );
    }

    /**
     * Unit test for class VLine.
     */

    public static void main( String[] argv )
    {
        Terminal terminal = new Terminal();

        terminal.println( "Self documenting unit test of VLine.");
        terminal.println( "The two Screens that follow should match.");
        terminal.println();
        terminal.println( "Hard coded picture:");
        terminal.println( "+++++++++");
        terminal.println( "+xx     +");
        terminal.println( "+xx     +");
        terminal.println( "+xx     +");
        terminal.println( "+xx *   +");
        terminal.println( "+xx *1  +");
        terminal.println( "+x  *   +");
```

```java
        terminal.println( "+x  *   +");
        terminal.println( "+   *   +");
        terminal.println( "+       +");
        terminal.println( "+++++++++");
        terminal.println();

        terminal.println( "Picture drawn using VLine methods:");
        Screen screen = new Screen( 7,  9 );

        Line vLine = new VLine(  7, 'x' );
        vLine.paintOn( screen );

        vLine.setLength(5);
        vLine.paintOn( screen, 1, 0 );

        vLine.setPaintChar('*');
        vLine.paintOn( screen, 3, 3 );

        vLine.setLength(1);
        vLine.setPaintChar('1');
        vLine.paintOn( screen, 4, 4 );

        screen.draw( terminal );
    }

}
```

```java
// joi/5/shapes/ShapeOnScreen.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

// This file is used in one of the Chapter 5 exercises on shapes.

/**
 * A ShapeOnScreen models a Shape to be painted at
 * a given position on a Screen.
 *
 * @see Shape
 * @see Screen
 *
 * @version 5
 */

public class ShapeOnScreen
{
    private Shape shape;
    private int x;
    private int y;

    /**
     * Construct a ShapeOncreen.
     *
     * @param shape the Shape
     * @param x its x coordinate
     * @param y its y coordinate
     */

    public ShapeOnScreen( Shape shape, int x, int y )
    {
        this.shape = shape;
        this.x     = x;
        this.y     = y;
    }

    /**
     * What Shape does this ShapeOnScreen represent?
     *
     * @return the Shape.
     */

    public Shape getShape() {
        return shape;
    }

    /**
     * The current x coordinate of this ShapeOnScreen.
     *
     * @return the x coordinate.
     */

    public int getX() {
        return x;
```

```java
    }

    /**
     * The current y coordinate of this ShapeOnScreen.
     *
     * @return the y coordinate.
     */

    public int getY() {
        return y;
    }

    /**
     * Unit test.
     */

    public static void main( String[] args ) {
        ShapeOnScreen sos = new ShapeOnScreen( null, 5, 7);
        System.out.println("Shape: " + sos.getShape());
        System.out.println("x:     " + sos.getX());
        System.out.println("y:     " + sos.getY());
    }
}
```

```java
// joi/5/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 5
 */

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

```java
        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( String owner )
    {
        this.owner = owner;
    }

    /**
     * The file’s owner.
     *
     * @return the owner of the file.
     */

    public String getOwner()
    {
        return owner;
    }

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

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

    /**
     * Set the modification date to "now".
     */

    protected void setModDate()
    {
        modDate = new Date();
    }

    /**
     * The date and time of the file’s last modification.
     *
     * @return the date and time of the file’s last modification.
     */

    public String getModDate()
    {
        return modDate.toString();
    }

    /**
     * The Directory containing this file.
     *
     * @return the parent directory.
     */

    public Directory getParent()
    {
        return parent;
    }
```

```java

    /**
     * A JFile whose parent is null is defined to be the root
     * (of a tree).
     *
     * @return true when this JFile is the root.
     */

    public boolean isRoot()
    {
        return (parent == null);
    }

    /**
     * How a JFile represents itself as a String.
     * That is,
     * <pre>
     *    owner   size    modDate    name+suffix
     * </pre>
     *
     * @return the String representation.
     */

    public String toString()
    {
        return getOwner() + "\t" +
            getSize()  + "\t" +
            getModDate()  + "\t" +
            getName() + getSuffix();
    }

    // Unit test: main() and static support

    private static Terminal terminal = new Terminal();

    /**
     * A unit test of JFile and its subclasses.
     */

    public static void main( String[] args )
    {
        out("Some hardwired, self documenting JFile system tests");
        out("create and then explore JFile hierarchy");
        out("   root                (owner sysadmin)");
        out("       billhome        (owner bill)");
        out("       ebhome          (owner eb)");
        out("           cs110       (owner eb)");
        out("               diary   (owner eb)");
        out("           insult      (owner bill)");
        Directory root   = new Directory( "",     "sysadmin", null );
        Directory home1  = new Directory( "ebhome",   "eb",   root );
        Directory home2  = new Directory( "billhome", "bill", root );

        TextFile  insult = new TextFile( "insult", "bill", home1,
                                         "Your mother wore sneakers,");
        insult.append( "\nin the shower." );
```

```java

       Directory cs110  = new Directory( "cs110", "eb", home1);
       cs110.addJFile( "diary",
                       new TextFile( "diary", "eb", cs110,
                                     "started work on Chapter 3"));

       out("\nlist contents of the root directory:" );
       list( root );

       out("\nlist contents of ebhome:" );
       list( home1 );

       out("\nretrieve billhome, list its contents (empty)):");
       list( (Directory) root.retrieveJFile("billhome") );

       out("\nretrieve insult, contents two line insult:");
       type( (TextFile)home1.retrieveJFile("insult"));

       out("\nretrieve file \"foo\" from ebhome, try to display it:");
       type( (TextFile)home1.retrieveJFile("foo") );

       out("\nlist contents of cs110 (one file):" );
       list( (Directory) home1.retrieveJFile("cs110") );

       out("path to root:\t "   + root.getPathName() );
       out("path to ebhome:\t " + home1.getPathName() );
       out("path to cs110:\t "  + cs110.getPathName() );
    }

    // display a listing of the contents of a Directory

    private static void list( Directory dir )
    {
        terminal.println( dir.getName() );
        terminal.println( dir.getSize() +
                          (dir.getSize() == 1
                          ? " file:" : " files:") );

        String[] fileNames = dir.getFileNames();
        for ( int i = 0; i < fileNames.length; i++ ) {
            String fileName = fileNames[i];
            JFile  jfile    = dir.retrieveJFile( fileName );
            terminal.println( jfile.toString() );
        }
    }

    // display the contents of a TextFile

    private static void type( TextFile file )
    {
       String whatToPrint;
       if (file == null) {
           whatToPrint = "no such file";
       }
       else {
           whatToPrint = file.getContents();
```

```java
        }
        terminal.println( whatToPrint );
    }

    // abbreviation for "terminal.println"

    private static void out( String s )
    {
        terminal.println( s );
    }
}
```

```java
// joi/5/jfiles/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 5
 */

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 Directo
     * @param creator  the owner of this new Directory
     * @param parent   the Directory in which this Directory lives.
     */

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

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

    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;
```

```java
    }

    /**
     * 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;
    }

    /**
     * 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] );
    }
}
```

```java
// joi/5/jfiles/TextFile.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * A TextFile is a JFile that holds text.
 *
 * @version 5
 */

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, String 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, String 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();
    }

    /**
```

```java
     * 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);
    }
}
```

```java
// joi/5/bank/Bank.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * A Bank object simulates the behavior of a simple bank/ATM.
 * It contains a Terminal object and a collection of
 * BankAccount objects.
 *
 * The visit method opens this Bank for business,
 * prompting the customer for input.
 *
 * To create a Bank and open it for business issue the command
 * <code>java Bank</code>.
 *
 * @see BankAccount
 * @version 5
 */

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
    private Month month;               // the current month.

    private TreeMap accountList;      // mapping names to accounts.

    // what the banker can ask of the bank

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

    // what the customer can ask of the bank

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

    /**
     * Construct a Bank with the given name and Terminal.
     *
     * @param bankName the name for this Bank.
     * @param atm  this Bank’s Terminal.
     */

    public Bank( String bankName, Terminal atm )
    {
        this.atm      = atm;
        this.bankName = bankName;
        accountList   = new TreeMap();
```

```java
        month         = new Month();
    }

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

    public void visit()
    {
        instructUser();

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

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


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

    private void openNewAccount()
    {
        String accountName = atm.readWord( "Account name: " );
        char accountType =
            atm.readChar( "Checking/Fee/Regular? (c/f/r): " );
        int startup = atm.readInt( "Initial deposit: " );
        BankAccount newAccount;
        switch( accountType ) {
        case ’c’:
            newAccount = new CheckingAccount( startup, this );
            break;
        case 'f':
            newAccount = new FeeAccount( 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 );
    }

    // 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( "c" ) ) {
                int amount = atm.readInt( "    amount of check: " );
                atm.println("    cashed check for " +
                            ((CheckingAccount)acct).honorCheck( 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());
            }
```

```java
            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()
    {
        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();
        // for each account
        //    account.newMonth()
    }

    // 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.

    private void instructUser()
    {
        atm.println( "Welcome to " + bankName );
        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();
    }

    /**
     * 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.
     */
```

```java

    public int getBalance()
    {
        return balance;
    }

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

    public int getNumberOfAccounts()
    {
        return accountList.size();
    }

    /**
     * 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>
     *
     * @param args the command line arguments:
     *         <pre>
     *         -e echo input.
     *         bankName any other command line argument.
     *         </pre>
     */

    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 = "Faithless Trust"; // 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();
    }
}
```

```java
// joi/5/bank/BankAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A BankAccount object has private fields to keep track
 * of its current balance, the number of transactions
 * performed and the Bank in which it is an account, and
 * and public methods to access those fields appropriately.
 *
 * @see Bank
 * @version 5
 */

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

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

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

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

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

    /**
     * Deposit the given amount, increasing this BankAccount’s
     * balance and the issuing Bank’s balance.
     * Counts as a transaction.
```

```java
     *
     * @param amount the amount to be deposited
     * @return amount deposited
     */

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

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

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

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

    public int getBalance()
    {
        return balance;
    }

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

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

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

    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;
    }

    /**
     * Action to take when a new month starts.
     */

    public abstract void newMonth();
}
```

```java
// joi/5/bank/RegularAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A RegularAccount is a BankAccount that has no special behavior.
 *
 * It does what a BankAccount does.
 */

public class RegularAccount extends BankAccount
{

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

    public RegularAccount( int initialBalance, Bank issuingBank )
    {
        super( initialBalance, issuingBank );
    }

    /**
     * Action to take when a new month starts.
     *
     * A RegularAccount does nothing when the next month starts.
     */

    public void newMonth() {
        // do nothing
    }

}
```

```java
// joi/5/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.
 *
 * @version 5
 */

public class CheckingAccount extends BankAccount
{
    private static int checkFee = 2;  // pretty steep for each check

    /**
     * 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.
     */

    public CheckingAccount( int initialBalance, Bank issuingBank )
    {
        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.
     */

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

    /**
     * Action to take when a new month starts.
     */

    public void newMonth()
    {
    }
}
```

```java
// joi/5/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.
 *
 * @version 5
 */

public class FeeAccount extends BankAccount
{
    private static int transactionFee = 1;

    /**
     * Constructor, accepting an initial balance and issuing Bank.
     *
     * @param initialBalance the opening balance.
     * @param issuingBank the bank that issued this account.
     */

    public FeeAccount( int initialBalance, Bank issuingBank )
    {
        super( initialBalance, issuingBank);
    }

    /**
     * The way a transaction is counted for a FeeAccount: it levies
     * a transaction fee as well as counting the transaction.
     */

    public void countTransaction()
    {
        incrementBalance( - transactionFee );
        super.countTransaction();
    }

    /**
     * Action to take when a new month starts.
     */

    public void newMonth()
    {
    }
}
```

```java
// joi/5/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 5
 */

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()
    {
        // needs completion
    }

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

//     public String toString()
//     {
//     }

```

```java
    /**
     * 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);
    }
}
```