```java
// joi/1/bank/BankAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A BankAccount object has a private field to keep track
 * of this account’s current balance, and public methods to
 * return and change the balance.
 *
 * @see Bank
 * @version 1
 */

public class BankAccount
{
    private int balance;  // work only in whole dollars

    /**
     * A constructor for creating a new bank account.
     *
     * @param initialBalance the opening balance.
     */

    public BankAccount( int initialBalance )
    {
        this.deposit( initialBalance );
    }

    /**
     * Withdraw the amount requested.
     *
     * @param amount the amount to be withdrawn.
     */

    public void withdraw( int amount )
    {
        balance = balance - amount;
    }

    /**
     * Deposit the amount requested.
     *
     * @param amount the amount to be deposited.
     */

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

    /**
     * The current account balance.
     *
     * @return the current balance.
     */
```

```java

    public int getBalance()
    {
        return balance;
    }
}
```

```java
// joi/1/bank/Bank.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A Bank object simulates the behavior of a simple bank/ATM.
 * It contains a Terminal object and two BankAccount objects.
 *
 * Its single public method is open, which 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 1
 */

public class Bank
{
    private String bankName;      // the name of this Bank

    private Terminal atm;         // for talking with the customer

    private BankAccount account1; // two accounts to play with
    private BankAccount account2;

    private static final int INITIAL_BALANCE = 200;
    private static final String HELPSTRING =
        "Transactions: exit, help, deposit, withdraw, balance";

    /**
     * Construct a Bank with the given name.
     * Create two new BankAccounts, each with a starting balance
     * of initialBalance.
     *
     * @param name the name of the Bank.
     */

    public Bank( String name )
    {
        bankName = name;
        atm      = new Terminal();
        account1 = new BankAccount( INITIAL_BALANCE );
        account2 = new BankAccount( INITIAL_BALANCE );
    }

    /**
     * Open the Bank for business.
     *
     * Send a whichAccount message prompting for a BankAccount
     * number, then send a processTransactionsForAccount
     * message to do the work.
     */

```

```java
    public void open()
    {
        atm.println( "Welcome to " + bankName );
        boolean bankIsOpen = true;
        while ( bankIsOpen ) {
            BankAccount account = this.whichAccount();
            if ( account == null ) {
                bankIsOpen = false;
            }
            else {
                this.processTransactionsForAccount(account);
            }
        }
        atm.println( "Goodbye from " + bankName );
    }

    // Prompt the user for an account number and return the
    // corresponding BankAccount object. Return null when
    // the Bank is about to close.

    private BankAccount whichAccount()
    {
        int  accountNumber =
            atm.readInt("Account number (1 or 2), 0 to shut down: ");

        if ( accountNumber == 1 ) {
            return account1;
        }
        else if ( accountNumber == 2 ) {
            return account2;
        }
        else if ( accountNumber == 0 ) {
            return null;
        }
        else {
            atm.println( "No account numbered " +
                                  accountNumber + "; try again" );
            return this.whichAccount();
        }
    }

    // Prompt the user for transaction to process.
    // Then send an appropriate message to account.

    private void processTransactionsForAccount( BankAccount account)
    {
        atm.println( HELPSTRING );

        boolean moreTransactions = true;
        while ( moreTransactions ) {
            String command = atm.readWord( "transaction: " );
            if ( command.equals( "exit" ) ) {
                moreTransactions = false;
            }
            else if ( command.equals( "help" ) ) {
                atm.println( HELPSTRING );
            }
            else if ( command.equals( "deposit" ) ) {
                int amount = atm.readInt( "amount: " );
                account.deposit( amount );
            }
            else if ( command.equals( "withdraw" ) ) {
                int amount = atm.readInt( "amount: " );
                account.withdraw( amount );
            }
            else if ( command.equals( "balance" ) ) {
                atm.println( account.getBalance() );
            }
            else{
                atm.println("sorry, unknown transaction" );
            }
        }
    }

    /**
     * The Bank simulation program begins here when the user
     * issues the command <code>java Bank</code>.
     *
     * @param args the command line arguments (ignored).
     */

    public static void main( String[] args )
    {
        Bank javaBank = new Bank( "Engulf and Devour" );
        javaBank.open();
    }
}
```

```java
// joi/1/lights/TrafficLight.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.awt.*;
import java.awt.event.*;

/**
 * A TrafficLight has three lenses: red, yellow and green.
 * It can be set to signal Go, Caution, Stop or Walk.
 *
 * @version 1
 */

public class TrafficLight extends Panel
{
    // Three Lenses and a Button

    private Lens red          = new Lens( Color.red );
    private Lens yellow       = new Lens( Color.yellow );
    private Lens green        = new Lens( Color.green );
    private Button nextButton = new Button("Next");

    /**
     * Construct a traffic light.
     */

    public TrafficLight()
    {
        this.setLayout(new BorderLayout());

        // create a Panel for the Lenses
        Panel lensPanel = new Panel();
        lensPanel.setLayout( new GridLayout( 3, 1 ) );
        lensPanel.add( red );
        lensPanel.add( yellow );
        lensPanel.add( green );
        this.add( BorderLayout.NORTH, lensPanel );

        // configure the "Next" button
        Sequencer sequencer = new Sequencer( this );
        NextButtonListener payAttention =
            new NextButtonListener( sequencer );
        nextButton.addActionListener( payAttention );
        this.add( BorderLayout.CENTER, nextButton);
    }

    // Methods that change the light

    /**
     * Set the light to stop (red).
     */

    public void setStop()
    {
```

```java
        red.turnOn();
        yellow.turnOff();
        green.turnOff();
    }

    /**
     * Set the light to caution (yellow).
     */

    public void setCaution()
    {
        red.turnOff();
        yellow.turnOn();
        green.turnOff();
    }

    /**
     * Set the light to go (green).
     */

    public void setGo()
    {
        red.turnOff();
        yellow.turnOff();
        green.turnOn();
    }

    /**
     * Set the light to walk.
     *
     * (In Boston, red and yellow signal walk.)
     */

    public void setWalk()
    {
        red.turnOn();
        yellow.turnOn();
        green.turnOff();
    }

    /**
     * The traffic light simulation starts at main.
     *
     * @param args ignored.
     */

    public static void main( String[] args )
    {
        Frame frame         = new Frame();
        TrafficLight light  = new TrafficLight();
        frame.add( light );
        frame.addWindowListener( new ShutDownLight() );
        frame.pack();
        frame.show();
    }

```

```
        // A ShutDownLight instance handles close events generated
        // by the underlying window system with its windowClosing
        // method.
        //
        // This is an inner class, declared inside the
        // TrafficLight class since it’s used only here.

       private static class ShutDownLight extends WindowAdapter
       {
            // Close the window by shutting down the light.

           public void windowClosing (WindowEvent e)
           {
               System.exit(0);
           }
       }
   }


```

```java
// joi/1/lights/NextButtonListener.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.awt.event.*;

/**
 * A NextButtonListener sends a "next" message to its
 * Sequencer each time a button to which it is listening
 * is pressed.
 *
 * @version 1
 */

public class NextButtonListener implements ActionListener
{
    private Sequencer sequencer;

    /**
     * Construct a listener that "listens for" a user’s
     * pressing the "Next" button.
     *
     * @param sequencer the Sequencer for the TrafficLight.
     */

    public NextButtonListener( Sequencer sequencer )
    {
        this.sequencer = sequencer;
    }

    /**
     * The action performed when a push of the button is detected:
     * send a next message to the Sequencer to advance it to
     * its next state.
     *
     * @param event the event detected at the button.
     */

    public void actionPerformed( ActionEvent event )
    {
        this.sequencer.next();
    }
}
```

```java
// joi/1/lights/Sequencer.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A Sequencer controls a TrafficLight. It maintains fields
 * for the light itself and the current state of the light.
 *
 * Each time it receives a "next" message, it advances to the
 * next state and sends the light an appropriate message.
 *
 * @version 1
 */

public class Sequencer
{
    // the TrafficLight this Sequencer controls
    private TrafficLight light;

    // represent the states by ints
    private final static int GO       = 0;
    private final static int CAUTION  = 1;
    private final static int STOP     = 2;

    private int currentState;

    /**
     * Construct a sequencer to control a TrafficLight.
     *
     * @param light the TrafficLight we wish to control.
     */

    public Sequencer( TrafficLight light )
    {
        this.light = light;
        this.currentState = GO;
        this.light.setGo();
    }

    /**
     * How the light changes when a next Button is pressed
     * depends on the current state. The sequence is
     * GO -> CAUTION -> STOP -> GO.
     */

    public void next()
    {
        switch ( currentState ) {

        case GO:
            this.currentState = CAUTION;
            this.light.setCaution();
            break;

        case CAUTION:
```

```java
            this.currentState = STOP;
            this.light.setStop();
            break;

        case STOP:
            this.currentState = GO;
            this.light.setGo();
            break;

        default: // This will never happen
            System.err.println("What color is the light?!");
        }
    }
}
```

```java
// joi/1/lights/Lens.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.awt.*;

/**
 * A Lens has a certain color and can either be turned on
 * (the color) or turned off (black).
 *
 * @version 1
 */

public class Lens extends Canvas
{
    private Color onColor;                 // color on
    private Color offColor = Color.black; // color off
    private Color currentColor;           // color the lens is now

    private final static int SIZE = 100;  // how big is this Lens?
    private final static int OFFSET = 20; // offset of Lens in Canvas

    /**
     * Construct a Lens to display a given color.
     *
     * The lens is black when it’s turned off.
     *
     * @param color the color of the lens when it is turned on.
     */

    public Lens( Color color )
    {
        this.setBackground( Color.black );
        this.onColor = color;
        this.setSize( SIZE , SIZE );
        this.turnOff();
    }

    /**
     * How this Lens paints itself.
     *
     * @param g a Graphics object to manage brush and color information.
     */

    public void paint( Graphics g )
    {
        g.setColor( this.currentColor );
        g.fillOval( OFFSET, OFFSET,
                    SIZE - OFFSET*2, SIZE - OFFSET*2 );
    }

    /**
     * Have this Lens display its color.
     */

```

```java
    public void turnOn()
    {
        currentColor = onColor;
        this.repaint();
    }

    /**
     * Darken this lens.
     */

    public void turnOff()
    {
        currentColor = offColor;
        this.repaint();
    }
}
```

```java
// joi/1/estore/EStore.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * An EStore object simulates the behavior of a simple on line
 * shopping web site.
 *
 * It contains a Terminal object to model the customer’s browser
 * and several Item objects a customer can add to her ShoppingCart.
 *
 * @version 1
 */

public class EStore
{
    private String storeName = "Virtual Minimal Minimall";

    // Use a Terminal object to communicate with customers.
    private Terminal browser = new Terminal();

    // The store stocks two kinds of Items.
    private Item widget = new Item(10); //  widgets cost $10
    private Item gadget = new Item(13); //  gadgets cost $13

    private String selectionList = "(gadget, widget, checkout)";

    /**
     * Visit this EStore.
     *
     * Loop allowing visitor to select items to add to her
     * ShoppingCart.
     */

    public void visit()
    {
        // Create a new, empty ShoppingCart.
        ShoppingCart basket = new ShoppingCart();

        // Print a friendly welcome message.
        browser.println( "Welcome to " + storeName );

        // Change to false when customer is ready to leave:
        boolean stillShopping = true;

        while ( stillShopping ) {
            Item nextPurchase = selectItem();
            if ( nextPurchase == null ) {
                stillShopping = false;
            }
            else {
                basket.add( nextPurchase );
            }
        }
        int numberPurchased = basket.getCount();
```

```java
        int totalCost       = basket.getCost();
        browser.println("We are shipping " + numberPurchased + " Items");
        browser.println("and charging your account $" + totalCost);
        browser.println("Thank you for shopping at " + storeName);
    }

    // Discover what the customer wants to do next:
    //  send browser a message to get customer input
    //  examine response to make a choice
    //  if response makes no sense give customer another chance

    private Item selectItem()
    {
        String itemName =
            browser.readWord("Item " + selectionList + ":");

        if ( itemName.equals("widget")) {
            return widget;
        }
        else if ( itemName.equals("gadget")) {
            return gadget;
        }
        else if ( itemName.equals("checkout" )) {
            return null;
        }
        else {
            browser.println( "No item named " +
                                  itemName + "; try again" );
            return selectItem();  // try again
        }
    }

    /**
     *  The EStore simulation program begins here when the user
     *  issues the command <code>java EStore</code>.
     */

    public static void main( String[] args )
    {
        // Print this to simulate delay while browser finds store
        System.out.println("connecting ...");

        // Create the EStore object.
        EStore webSite = new EStore();

        // Visit it.
        webSite.visit();
    }
} // end of class EStore
```

```java
// joi/1/estore/Item.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * An Item models an object that might be stocked in a store.
 * Each Item has a cost.
 *
 * @version 1
 */

public class Item
{
    private int cost;

    /**
     *  Construct an Item object.
     *
     *  @param itemCost the cost of this Item.
     */

    public Item( int itemCost )
    {
        cost = itemCost;
    }

    /**
     * How much does this Item cost?
     *
     * @return the cost.
     */

    public int getCost()
    {
        return cost;
    }
}
```

```java
// joi/1/estore/ShoppingCart.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A ShoppingCart keeps track of a customer’s purchases.
 *
 * @see EStore
 * @version 1
 */

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

    /**
     *  Construct a new empty ShoppingCart.
     */

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

    /**
     * When this ShoppingCart is asked to add an Item to itself
     * it updates its count field and then updates its cost
     * field by sending the Item a getCost message.
     *
     * @param purchase the Item being added to this ShoppingCart.
     */

    public void add( Item purchase )
    {
        count++; // Java idiom for count = count + 1;
        cost = cost + purchase.getCost();
    }

    /**
     * What happens when this ShoppingCart is asked how many
     * Items it contains.
     *
     * @return the count of Items.
     */

    public int getCount()
    {
        return count;
    }

    /**
     * What happens when this ShoppingCart is asked the total
     * cost of the Items it contains.
```

```java
     *
     * @return the total cost.
     */

    public int getCost()
    {
        return cost;
    }
}
```

```java
// joi/2/change/Change.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * Program to make change.
 * Uses the Terminal method readInt() for prompted integer input.
 *
 * @version 2
 */

public class Change
{
    /**
     * Illustrate simple arithmetic.
     */

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

        amount      = terminal.readInt("Amount, in cents: ");
        int dimes   = amount/10;
        amount      = amount % 10;
        int nickels = amount / 5;
        amount      = amount % 5;
        terminal.println(dimes   + " dimes");
        terminal.println(nickels + " nickels");
        terminal.println(amount  + " pennies");
    }
}
```

```java
// joi/2/linear/Temperatures.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * Temperature conversion program,
 * for exercising LinearEquation objects.
 *
 * @version 2
 */

public class Temperatures
{
    /**
     * First a hardcoded test of Celsius-Fahrenheit conversion,
     * then a loop allowing the user to test interactively.
     */

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

        // create a Celsius to Fahrenheit converter
        LinearEquation c2f = new LinearEquation( 9.0/5.0, 32.0 );

        // ask it to tell us its inverse, for F to C
        LinearEquation f2c = c2f.getInverse();

        ///////////////////////////////////////////////////
        // Testing style 1: Hard coded, self-documenting //
        ///////////////////////////////////////////////////

        terminal.println( "Hard coded self documenting tests:" );
        terminal.print( "c2f.compute( 0.0 ), should see 32.0: ");
        terminal.println( c2f.compute( 0.0 ) );
        terminal.print( "f2c.compute( 212.0 ), should see 100.0: ");
        terminal.println( f2c.compute( 212.0 ) );

        //////////////////////////////////
        // Testing style 2: Interactive //
        //////////////////////////////////

        terminal.println();
        terminal.println( "Interactive tests:" );
        while ( terminal.readYesOrNo("more?") ) {
            double degreesCelsius  =
                terminal.readDouble( "Celsius: " );
            terminal.println(" = "
                             + c2f.compute( degreesCelsius )
                             + " degrees Fahrenheit" );
            double degreesFahrenheit  =
                terminal.readDouble( "degrees Fahrenheit: " );
            terminal.println(" = "
                             + f2c.compute( degreesFahrenheit )
                             + " degrees Celsius" );
```

```java
        }
    }
}
```

```java
// joi/2/linear/LinearEquation.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A LinearEquation models equations of the form y = mx + b.
 *
 * @version 2
 */

public class LinearEquation
{
    private double m;      // The equations’s slope
    private double b;      // The equations’s y-intercept

    /**
     * Construct a LinearEquation from a slope and y-intercept.
     *
     * @param m the slope.
     * @param b the y-intercept.
     */

    public LinearEquation( double m, double b )
    {
        this.m = m;
        this.b = b;
    }

    /**
     * Construct a LinearEquation from two points.
     *
     * @param x1 the x coordinate of the first point
     * @param y1 the y coordinate of the first point
     * @param x2 the x coordinate of the second point
     * @param y2 the y coordinate of the second point
     */

    public LinearEquation( double x1, double y1,
                           double x2, double y2 )
    {
        m = (y2 - y1) / (x2 - x1);
        b = y1 - x1 * m;
    }

    /**
     * Compute y, given x.
     *
     * @param x the input value.
     * @return the corresponding value of y: mx+b.
     */

    public double compute( double x )
    {
        return m*x + b;
    }
```

```java

    /**
     * Compute the inverse of this linear equation.
     *
     * @return the LinearEquation object you get by "solving for x".
     */

    public LinearEquation getInverse()
    {
        return new LinearEquation( 1.0/m, -b/m );
    }
}
```

```java
// joi/2/arithmetic/IntArithmetic.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * Interactive play with integer arithmetic in Java,
 * using a Terminal for input and output.
 */

public class IntArithmetic
{
    private static Terminal terminal = new Terminal();

    /**
     * main prompts for pairs of numbers to add and to divide
     * until the bored user decides to quit.
     */

    public static void main(String[] args)
    {
        while ( terminal.readYesOrNo( "Try int z = x + y ? " ) ) {
            tryIntegerAddition( );
        }
        while ( terminal.readYesOrNo( "Try int z = x / y ? " ) ) {
            tryIntegerDivision();
        }
    }

    // Prompt for two ints and add them.

    private static void tryIntegerAddition()
    {
        int x = terminal.readInt( "x = ");
        int y = terminal.readInt( "y = " );
        terminal.println( "z = " + (x+y) );
    }

    // Prompt for two ints and divide the first by
    // the second.

    private static void tryIntegerDivision()
    {
        int x = terminal.readInt( "x = ");
        int y = terminal.readInt( "y = " );
        terminal.println( "z = " + (x/y) );
    }
}
```

```java
// joi/3/textfiles/TextFile.java
//
//
// 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();
    }

```

```java
    /**
     * 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 + '\n' + 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();
    }

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

    public String getOwner()
    {
        return owner;
    }

    /**
     * A definition of main(), used only for testing this class.
     *
     * Executing
     * <pre>
     *     %> java TextFile
     * </pre>
     * produces the output:
     * <pre>
     * TextFile myTextFile contains 13 characters.
     * Created by bill, Sat Dec 29 14:02:37 EST 2001
     * Hello, world.
     *
     * append new line "How are you today?"
     * Hello, world.
     * How are you today?
     * TextFile myTextFile contains 32 characters.
     * Modified Sat Dec 29 14:02:38 EST 2001
     * </pre>
     */

    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() );
    }
}
```

```java
// joi/3/shapes/DemoShapes.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A short demonstration program for HLine and Box.
 *
 * @version 3
 */

public class DemoShapes
{
    /**
     * Paint some shapes on a Screen and draw it to a Terminal.
     */

    public static void main( String[] args )
    {
        Terminal t = new Terminal();
        Screen   s = new Screen( 36, 12 );

        HLine h1 = new HLine( 10, 'R' );
        Box   b1 = new Box( 5, 6, 'G' );
        Box   b2 = new Box( 5, 6, 'B' );

        h1.paintOn( s ); // at position (0,0)
        b1.paintOn( s, 2, 2 );
        b2.paintOn( s, 4, 5 );

        t.println( "A Screen with an HLine and two Boxes:" );
        s.draw( t );
    }
}
```

```java
// joi/3/shapes/HLine.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A horizontal line has a length and a paintChar used
 * used to paint the line on a Screen.
 *
 * @version 3
 */

public class HLine
{
    private int  length;      // length in (character) pixels.
    private char paintChar;   // character used for painting.

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

    public HLine( int length, char paintChar )
    {
        this.length = length;
        this.paintChar = paintChar;
    }

    /**
     * Paint this HLine 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 void paintOn( Screen s, int x, int y )
    {
        for ( int i = 0; i < length; i = i+1 ) {
            s.paintAt( paintChar, x+i , y );
        }
    }

    /**
     * Paint this HLine 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
    /**
     * 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;
    }

    /**
     * Unit test for class HLine,
     * assuming Screen and Terminal work.
     */

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

        terminal.println( "Unit test of HLine.");
        terminal.println( "You should see this Screen twice: " );
        terminal.println( "++++++++++++++++++++++");
        terminal.println( "+xxxxxxxxxx          +");
        terminal.println( "+xxxxx               +");
        terminal.println( "+                    +");
        terminal.println( "+   *****            +");
        terminal.println( "+    1               +");
        terminal.println( "+                    +");
        terminal.println( "++++++++++++++++++++++");
        terminal.println( "");

        Screen   screen   = new Screen( 20, 6 );

        HLine hline1 = new HLine( 10, ’x’ );
        HLine hline2 = new HLine(  5, ’*’ );
        HLine hline3 = new HLine(  1, ’1’ );

        hline1.paintOn( screen );
        hline1.setLength(5);
        hline1.paintOn( screen, 0, 1 );
        hline2.paintOn( screen, 3, 3 );
        hline3.paintOn( screen, 4, 4 );

        screen.draw( terminal );
    }
}
```

```java
// joi/3/shapes/Box.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A Box has a width, a height and a paintChar used
 * used to paint the Box on a Screen.
 *
 * Examples:
 * <pre>
 *  new Box( 3, 4, ’G’ )   new Box( 1, 1, ’$’ )
 *
 *         GGG                     $
 *         GGG
 *         GGG
 *         GGG
 * </pre>
 *
 * @version 3
 */

public class Box
{
    private int  width;      // width  in (character) pixels
    private int  height;     // height in (character) pixels
    private char paintChar;  // character used for painting

    /**
     * Construct a box.
     *
     * @param width width in (character) pixels.
     * @param height height in (character) pixels.
     * @param paintChar character used for painting this Box.
     */

    public Box( int width, int height, char paintChar )
    {
        this.width     = width;
        this.height    = height;
        this.paintChar = paintChar;
    }

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

    public void paintOn( Screen s, int x, int y )
    {
        HLine hline = new HLine( width, paintChar );
        for ( int i = 0; i < height; i++ ) {
            hline.paintOn( s, x, y+i );
```

```java
        }
    }

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

    public void paintOn( Screen s )
    {
        paintOn( s, 0, 0 );  // or this.paintOn(s,0,0);
    }

    /**
     * Get the width of this Box.
     *
     * @return width of box (expressed as a number
     *         of characters).
     */

    public int getWidth()
    {
        return width;
    }

    /**
     * Get the height of this Box.
     *
     * @return the height in (character) pixels.
     */

    public int getHeight()
    {
        return height;
    }

    /**
     * Set the width of this Box.
     *
     * @param width the new width in (character) pixels.
     */

    public void setWidth( int width )
    {
        this.width = width;
    }

    /**
     * Set the height of this Box.
     *
     * @param height the new height in (character) pixels.
     */

    public void setHeight( int height )
    {
        this.height = height;
    }

    /**
     * Unit test for class Box,
     * assuming Screen and Terminal work.
     */

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

        terminal.println( "Unit test of Box.");
        terminal.println( "You should see this Screen twice: " );
        terminal.println( "++++++++++++++++++++++");
        terminal.println( "+RRRR                +");
        terminal.println( "+RRRR                +");
        terminal.println( "+RRGGG               +");
        terminal.println( "+RRGGG               +");
        terminal.println( "+RRGGG               +");
        terminal.println( "+  GGRRRRRRR         +");
        terminal.println( "++++++++++++++++++++++");
        terminal.println();

        Screen screen   = new Screen( 20, 6 );

        Box box1 = new Box( 4, 5, 'R' );
        Box box2 = new Box( 3, 4, 'G' );

        box1.paintOn( screen );
        box2.paintOn( screen, 2, 2 );

        // test reference model for objects
        box2 = box1;
        int oldWidth = box2.getWidth();
        box1.setWidth(oldWidth+3);
        box2.paintOn( screen, 4, 5 );

        screen.draw( terminal );
    }
}
```

```java
// joi/3/shapes/TestShapes.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A program to test shapes.
 *
 * @version 3
 */

class TestShapes
{
    /**
     * Paint shapes on a Screen and draw it to a Terminal.
     */

    public static void main( String[] argv )
    {
        Terminal t = new Terminal();
        Screen s;

        t.println( "An empty 10 x 3 Screen:" );
        s = new Screen( 10, 3 );
        s.draw( t );

        t.println( "A 20 x 10 Screen with 3 HLines:" );
        s = new Screen( 20, 10 );
        HLine h1 = new HLine( 10, 'R' );
        HLine h2 = new HLine( 15, 'G' );

        h1.paintOn( s, 0, 0 );
        h2.paintOn( s, 0, 1 );
        (new HLine( 15, 'B' )).paintOn( s, 0, 2 ); // tricky to read
        s.draw( t );

        t.println( "Clear that screen,");
        s.clear();

        t.println( "draw 3 Boxes (2 overlapping):" );
        Box   b = new Box( 6, 5, 'R' );
        b.paintOn( s, 1, 1 );
        b = new Box( 7, 4, 'G'); // create a new (different) Box b
        b.paintOn( s, 2, 3 );    // paint Box b on s
        b.paintOn( s, 17, 5 );   // paint Box b partly off the Screen
        s.draw( t );
    }
}
```

```java
// joi/3/shapes/InteractiveShapes.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * Interactive program to study shapes.
 *
 * @version3
 */

public class InteractiveShapes
{
    public static void main( String[] args )
    {
        Terminal t = new Terminal();
        Screen s = new Screen(
                              t.readInt("screen width:  "),
                              t.readInt("screen height: "));
        char c = 'a';
        int x,y;
        while ( t.readYesOrNo("more")) {
            char shape = t.readChar("h(line), b(ox), c(lear): ");
            switch (shape) {
            case 'h':
                int length = t.readInt("HLine length: ");
                x          = t.readInt("x coordinate: ");
                y          = t.readInt("y coordinate: ");
                (new HLine(length, c++)).paintOn(s,x,y);
                break;
            case 'b':
                int w = t.readInt("Box width: ");
                int h = t.readInt("Box height: ");
                x     = t.readInt("x coordinate: ");
                y     = t.readInt("y coordinate: ");
                (new Box(w,h,c++)).paintOn(s,x,y);
                break;
            case 'c':
                s.clear();
                break;
            default:
                t.println("try again");
                continue;
            }
            s.draw(t);
        }
    }
}
```

```java
// joi/3/shapes/TextLine.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

// This file contains stubs for the methods.

/**
 * A horizontal line of character text.
 *
 * @version 3
 */

public class TextLine
{
    /**
     * Construct a TextLine.
     *
     * @param text the text of the line.
     */

    public TextLine( String text )
    {
    }

    /**
     * Paint this TextLine 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 void paintOn( Screen s, int x, int y )
    {
    }

    /**
     * Draw the TextLine to 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 );
    }

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

    public int getLength()
    {
```

```java
        return 0; // replace with the right answer
    }

    /**
     * Unit test for class.TextLine,
     * assuming Screen and Terminal work.
     */

    public static void main( String[] args )
    {
    }
}
```

```java
// joi/3/shapes/Screen.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A Screen is a (width*height) grid of (character) 'pixels'
 * on which we may paint various shapes.  It can be drawn to
 * a Terminal.
 *
 * @version 3
 */

public class Screen
{
    /**
     * The character used to paint the screen's frame.
     */

    private static final char FRAMECHAR = '+';
    private static final char BLANK = ' ';
    private int width;
    private int height;
    private char[][] pixels;

    /**
     * Construct a Screen.
     *
     * @param width the number of pixels in the x direction.
     * @param height the number of pixels in the y direction.
     */

    public Screen( int width, int height )
    {
        this.width  = width;
        this.height = height;
        pixels = new char[width][height];
        clear();
    }

    /**
     * Clear the Screen, painting a blank at every pixel.
     */

    public void clear()
    {
        for (int x  = 0; x < width; x++) {
            for ( int y = 0; y < height; y++ ) {
                pixels[x][y] = BLANK;
            }
        }
    }

    /**
     * Paint a character pixel at position (x,y).
     *
```

```java
     * @param c the character to be painted.
     * @param x the (horizontal) x position.
     * @param y the (vertical) y position.
     */

    public void paintAt( char c, int x, int y )
    {
        if ( 0 <= x && x < width &&
             0 <= y && y < height) {
            pixels[x][y] = c;
        }
        // Otherwise off the Screen - nothing is painted.
    }

    /**
     * How wide is this Screen?
     *
     * @return the width.
     */

    public int getWidth()
    {
        return width;
    }

    /**
     * How high is this Screen?
     *
     * @return the height.
     */

    public int getHeight()
    {
        return height;
    }

    /**
     * Draw this Screen on a Terminal.
     *
     * @param t the Terminal on which to draw this Screen.
     */

    public void draw( Terminal t )
    {
        for ( int col = -1; col < width+1 ; col++ ) {  // top edge
            t.print(FRAMECHAR);
        }
        t.println();
        for ( int row = 0; row < height; row++ ) {
            t.print(FRAMECHAR);                      // left edge
            for ( int col = 0; col < width; col++ ) {
                t.print( pixels[col][row] );
            }
            t.println( FRAMECHAR );                  // right edge
        }
        for ( int col = -1; col < width+1 ; col++ ) {  // bottom edge
```

```
            t.print(FRAMECHAR);
        }
        t.println();
    }
}
```

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

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

/// import java.util.??

/**
 * A Bank object simulates the behavior of a simple bank/ATM.
 * It contains a Terminal object and a collection of
 * BankAccount objects.
 *
 * Its public method visit opens this Bank for business,
 * prompting the customer for input.
 *
 * To create a Bank and open it for business issue the command
 * <code>java Bank</code>.
 *
 * @see BankAccount
 * @version 4
 */

public class Bank
{
    private String bankName;           // the name of this Bank
    private Terminal atm;              // for talking with the customer
    private int balance = 0;           // total cash on hand
    private int transactionCount = 0;  // number of Bank transactions don

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

    // what the banker can ask of the bank

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

    // what the customer can ask of the bank

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

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

    public Bank( String bankName, Terminal atm )
    {
```

```java
        this.atm      = atm;
        this.bankName = bankName;
        // initialize collection:
        accountList   = new BankAccount[NUM_ACCOUNTS]; ///

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

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

    public void visit()
    {
        instructUser();

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

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


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

    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\n");
        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)));
                }
            }
```

```java
            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( "\nSummaries of individual accounts:" );
        atm.println( "account  balance   transaction count" );
        for (int i = 0; i < NUM_ACCOUNTS; i++ ) {              ///
            atm.println(i + "\t" + accountList[i].getBalance() +     ///
                        "\t" + accountList[i].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.
     */

    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 ...
    }

    /**
     * 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 = "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();
    }
}
```

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

/**
 * A BankAccount object has private fields to keep track
 * of its current balance, the number of transactions
 * performed and the Bank in which it is an account, and
 * and public methods to access those fields appropriately.
 *
 * @see Bank
 * @version 4
 */

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

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

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

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

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

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

```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 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;
    }
}
```

```
open
1000
open
2000
help
report
open
3000
customer
0
balance
deposit
9999
balance
quit
customer
1
transfer
9
transfer
2
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:
account  balance   transaction count
0       $1000           1
1       $2000           1

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

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

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

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

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

banker command: exit

```

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

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

Goodbye from River Bank
```

```
open
groucho
1000
customer
harpo
open
harpo
2000
help
report
open
chico
3000
customer
groucho
balance
deposit
9999
balance
quit
customer
harpo
transfer
chico
45
quit
exit
```

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

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

banker command: report

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

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

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

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

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

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

```

banker command: exit

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

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

Goodbye from River Bank
```

```java
// joi/examples/Reverse.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.ArrayList;

/**
 * Reverse the order of lines entered from standard input.
 */

public class Reverse
{

    /**
     * Read lines typed at the terminal until end-of-file,
     * saving them in an ArrayList.
     *
     * Then print the lines in reverse order.
     */

    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 );
        }
    }
}
```

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

import java.util.*;

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

public class Dictionary
{
    private TreeMap entries;

    /**
     * Construct an empty Dictionary.
     */

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

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

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

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

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

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

```java
     */

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

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

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

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

/**
 * 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;
    }
}
```

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

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

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

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

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

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

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

```

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

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

        // look up some words
        String word;

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

        // words entered interactively
        t.println("\nlook up words, \"quit\" to quit");
        while (true) {
            word = t.readWord("word> ");
            if (word.equals("quit")) {
                break;
            }
            printDefinition(word);
        }
    }
}
```

```
// joi/3/textfiles/TextFile.java
//
//
// 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 + ’\n’ + 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();
```

```java
    }

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

    public String getOwner()
    {
        return owner;
    }

    /**
     * A definition of main(), used only for testing this class.
     *
     * Executing
     * <pre>
     *     %> java TextFile
     * </pre>
     * produces the output:
     * <pre>
     * TextFile myTextFile contains 13 characters.
     * Created by bill, Sat Dec 29 14:02:37 EST 2001
     * Hello, world.
     *
     * append new line "How are you today?"
     * Hello, world.
     * How are you today?
     * TextFile myTextFile contains 32 characters.
     * Modified Sat Dec 29 14:02:38 EST 2001
     * </pre>
     */

    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() );
```

```java
        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() );
    }
}
```

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

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

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

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

    public Directory( )
    {
    }

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

    public int getSize()
    {
        return 0;
    }

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

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

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

    public TextFile retrieveTextFile( String filename )
    {
```

```java
        return null;
    }

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

        browser.println( "Currently available:");
        catalog.show(browser);
        while ( true ) { // loop forever ...
            String nextPurchase = browser.readWord(
                   "select your purchase, checkout, help: ");

            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 " +
                        basket.getCount() + " 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 "-e" instantiate a
     *  Terminal that echoes its input.
     *
     *  The next command line argument (if there is one)
     *  is the name of the EStore.
     *
     *  @param args <-e> <storeName>
     */
    public static void main( String[] args )
    {

        String storeName = "Virtual Minimal Minimall"; //default

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

        // if first argument was "-e" then look at second for store name
        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();
    }
}
```

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

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

public class ShoppingCart
{
    /// replace these two fields by a single ArrayList
    private int count; // number of Items in this ShoppingCart
    private int cost;  // total cost of Items in this ShoppingCart

    /**
     * Construct a new empty ShoppingCart.
     */

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

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

    public void addItem( Item item )
    {
        /// this code just keeps track of the totals
        /// replace it with code that adds the item to the list
        count++;
        this.cost += item.getCost(); // Java idiom: a += b means a = a +
    }

    /**
     * Return an Item from this ShoppingCart.
     *
     * @param item the Item to return.
     */

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

    /**
     * What happens when this ShoppingCart is asked how many
```

```java
     * Items it contains.
     *
     * @return the number of items in this ShoppingCart.
     */

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

    /**
     * What happens when this ShoppingCart is asked the total
     * cost of the Items it contains.
     *
     * @return the total cost of the items in this ShoppingCart.
     */
    public int getCost()
    {
        /// get this information from the list,
        /// since the cost field no longer exists
        return cost;
    }

    /**
     * Write the contents of this ShoppingCart to a Terminal.
     *
     * @param t the Terminal to use for output.
     */

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

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

/**
 * An Item models an object that might be stocked in a store.
 * Each Item has a cost.
 *
 * @version 4
 */

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

    /**
     *  Construct an Item object.
     *
     *  @param name the nme of this Item.
     *  @param cost the cost of this Item.
     */

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

    /**
     * How much does this Item cost?
     *
     * @return the cost.
     */

    public int getCost()
    {
        return cost;
    }

    /**
     * What is this Item called?
     *
     * @return the name.
     */

    public String getName()
    {
        return name;
    }
}
```

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

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.
     *
```

```java
     * @param t the Terminal to print to.
     */

    public void show( Terminal t )
    {
        // loop on items, printing name and cost
        t.println("  [sorry, can’t yet print Catalog contents]");
    }
}
```

```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." );

        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);
    }
}
```

```java
//  joi/6/juno/Juno.java
//
//
//  Copyright 2003 Bill Campbell and Ethan Bolker

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

/**
 * Juno (Juno's Unix NOt) mimics a command line operating system
 * like Unix.
 * <p>
 * A Juno system has a name, a set of Users, a JFile system,
 * a login process and a set of shell commands.
 *
 * @see User
 * @see JFile
 * @see ShellCommand
 *
 * @version 6
 */

public class Juno
{
    private final static String os      = "Juno";
    private final static String version = "6";

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

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

    private ShellCommandTable commandTable; // shell commands

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

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

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

        // the file system

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

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

        // create, then start a command line login interpreter

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

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

    public String getHostName()
    {
        return hostName;
    }

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

    public String getOS()
    {
        return os;
    }

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

    public String getVersion()
    {
        return version;
    }

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

    public Directory getUserHomes()
    {
        return userHomes;
    }

    /**
     * The shell command table for this system.
     *
     * @return the shell command table.
     */

    public ShellCommandTable getCommandTable()
    {
        return commandTable;
    }

    /**
     * Look up a user by user name.
     *
     * @param username the user’s name.
     * @return the appropriate User object.
     */

    public User lookupUser( String username )
    {
        return (User) users.get( username );
    }

    /**
     * Create a new User.
     *
     * @param userName the User’s login name.
     * @param home her home Directory.
     * @param realName her real name.
     * @return newly created User.
     */

    public User createUser( String userName, Directory home,
                            String realName )
    {
        User newUser = new User( userName, home, realName );
        users.put( userName, newUser );
        return newUser;
    }

    /**
     * The Juno system may be given the following command line
     * arguments.
     * <pre>
     *
     * -e:         Echo all input (useful for testing).
     *
     * -version:   Report the version number and exit.
     *
     * [hostname]: The name of the host on which
     *             Juno is running (optional).
     * </pre>
     */

    public static void main( String[] args )
```

```java
    {
        // Parse command line options

        boolean echoInput = false;
        String  hostName  = "mars";

        for (int i=0; i < args.length; i++) {
            if (args[i].equals("-version")) {
                System.out.println( os + " version " + version );
                System.exit(0);
            }
            if (args[i].equals("-e")) {
                echoInput = true;
            }
            else {
                hostName = args[i];
            }
        }

        // create a Juno instance, which will start itself

        new Juno( hostName, echoInput );
    }
}
```

```
// joi/6/juno/LoginInterpreter.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Interpreter for Juno login commands.
 *
 * There are so few commands that if-then-else logic is OK.
 *
 * @version 6
 */

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

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

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

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

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

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

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

    public void CLILogin()
    {
```

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

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

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

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

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

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

    private void welcome()
    {
        console.println( "Welcome to " + system.getHostName() +
                         " running " + system.getOS() +
                         " version " + system.getVersion() );
        help();
    }

    // Remind user of available commands.

    private void help()
    {
        console.println( LOGIN_COMMANDS );
        console.println("");
    }
}
```

```java
// joi/6/juno/Shell.java
//
//
// Copyright 2003, Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Models a shell (command interpreter)
 *
 * The Shell knows the (Juno) system it’s working in,
 * the User who started it,
 * and the console to which to send output.
 *
 * It keeps track of the the current working directory (.) .
 *
 * @version 6
 */

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

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

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

    // Run the command line interpreter

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

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

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

    private boolean interpret( String inputLine )
    {
        StringTokenizer st = stripComments(inputLine);
        if (st.countTokens() == 0) {
            return true;             // skip blank line
        }
        String commandName = st.nextToken();
        if (commandName.equals( "logout" )) {
            return false;            // user is done
        }
        ShellCommand commandObject =
            system.getCommandTable().lookup( commandName );
        if (commandObject == null ) {
            console.errPrintln( "Unknown command: " + commandName );
        }
        else {
            commandObject.doIt( st, this );
        }
        return true;
    }

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

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

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

    public String getPrompt()
    {
        return system.getHostName() + "> ";
    }

    /**
     * The User associated with this Shell.
     *
     * @return the user.
     */

    public User getUser()
    {
        return user;
    }

    /**
     * The current working directory for this Shell.
     *
     * @return the current working directory.
     */

    public Directory getDot()
    {
        return dot;
    }

    /**
     * Set the current working directory for this Shell.
     *
     * @param dot the new working directory.
     */

    public void setDot(Directory dot)
    {
        this.dot = dot;
    }

    /**
     * The console associated with this Shell.
     *
     * @return the console.
     */

    public Terminal getConsole()
    {
        return console;
    }

    /**
     * The Juno object associated with this Shell.
     *
     * @return the Juno instance that created this Shell.
     */

    public Juno getSystem()
    {
        return system;
    }
}
```

```java
// joi/6/juno/ShellCommand.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Model those features common to all ShellCommands.
 *
 * Each concrete extension of this class provides a constructor
 * and an implementation for method doIt.
 *
 * @version 6
 */

public abstract class ShellCommand
{
    private String helpString;   // documents the command
    private String argString;    // any args to the command

    /**
     * A constructor, always called (as super()) by the subclass.
     * Used only for commands that have arguments.
     *
     * @param helpString a brief description of what the command does.
     * @param argString a prototype illustrating the required arguments.
     */

    protected ShellCommand( String helpString, String argString )
    {
        this.argString  = argString;
        this.helpString = helpString;
    }

    /**
     * A constructor for commands having no arguments.
     *
     * @param helpString a brief description of what the command does.
     */

    protected ShellCommand( String helpString )
    {
        this( helpString, "" );
    }

    /**
     * Execute the command.
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     */

    public abstract void doIt( StringTokenizer args, Shell sh );

    /**
```

```java
     * Help for this command.
     *
     * @return the help string.
     */

    public String getHelpString()
    {
        return helpString;
    }

    /**
     * The argument string prototype.
     *
     * @return the argument string prototype.
     */

    public String getArgString()
    {
        return argString;
    }
}
```

```java
// joi/6/juno/MkdirCommand.java
//
//
// Copyright 2003, Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * The Juno shell command to create a new directory.
 * Usage:
 * <pre>
 *     mkdir directory-name
 * </pre>
 *
 * @version 6
 */

public class MkdirCommand extends ShellCommand
{
    /**
     * Construct a MkdirCommand object.
     */

    public MkdirCommand()
    {
        super( "create a subdirectory of the current directory",
               "directory-name" );
    }

    /**
     * Create a new Directory in the current Directory.
     *
     * @param args the remainder of the command line.
     * @param sh the current shell
     */

    public void doIt( StringTokenizer args, Shell sh )
    {
        String filename = args.nextToken();
        new Directory( filename, sh.getUser(), sh.getDot() );
    }
}
```

```java
// joi/6/juno/TypeCommand.java
//
//
// Copyright 2003, Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * The Juno shell command to display the contents of a
 * text file.
 * Usage:
 * <pre>
 *     type textfile
 * </pre>
 *
 * @version 6
 */

public class TypeCommand extends ShellCommand
{
    /**
     * Construct a TypeCommand object.
     */

    TypeCommand()
    {
        super( "display contents of a TextFile", "textfile" );
    }

    /**
     * Display the contents of a TextFile.
     *
     * @param args the reminder of the command line.
     * @param sh the current Shell
     */

    public void doIt( StringTokenizer args, Shell sh )
    {
        String filename = args.nextToken();
        sh.getConsole().println(
            ( (TextFile) sh.getDot().
              retrieveJFile( filename ) ).getContents() );
    }
}
```

```java
// joi/6/juno/HelpCommand.java
//
//
// Copyright 2003, Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * The Juno shell command to display help on the shell commands.
 * Usage:
 * <pre>
 *     help
 * </pre>
 *
 * @version 6
 */

public class HelpCommand extends ShellCommand
{
    /**
     * Construct a HelpCommand object.
     */

    HelpCommand()
    {
        super( "display ShellCommands" );
    }

    /**
     * Display help for all commands.
     *
     * @param args the remainder of the command line.
     * @param sh the current shell
     */

    public void doIt( StringTokenizer args, Shell sh )
    {
        // Get command keys from global table, print them out,
        // followed by command’s help string.

        sh.getConsole().println( "shell commands" );
        ShellCommandTable table = sh.getSystem().getCommandTable();
        String[] names = table.getCommandNames();
        for (int i = 0; i < names.length; i++ ) {
            String cmdname = names[i];
            ShellCommand cmd = table.lookup( cmdname );
            sh.getConsole().
                println( "  " + cmdname + ": " + cmd.getHelpString() );
        }
    }
}
```

```java
// joi/6/juno/NewfileCommand.java
//
//
// Copyright 2003, Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * The Juno shell command to create a text file.
 * Usage:
 * <pre>
 *     newfile filename contents
 * </pre>
 *
 * @version 6
 */

public class NewfileCommand extends ShellCommand
{
    /**
     * Construct a NewfileCommand object.
     */

    public NewfileCommand()
    {
        super( "create a new TextFile", "filename contents" );
    }

    /**
     * Create a new TextFile in the current Directory.
     *
     * @param args the remainder of the command line.
     * @param sh the current shell
     */

    public void doIt( StringTokenizer args, Shell sh )
    {
        String filename = args.nextToken();
        String contents = args.nextToken("").trim(); // rest of line
        new TextFile( filename, sh.getUser(), sh.getDot(), contents );
    }
}
```

```
// joi/6/juno/ShellCommandTable.java (version 6)
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * A ShellCommandTable object maintains a dispatch table of
 * ShellCommand objects keyed by the command names used to invoke
 * them.
 *
 * To add a new shell command to the table, install it from
 * method fillTable().
 *
 * @see ShellCommand
 *
 * @version 6
 */

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

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

    public ShellCommandTable()
    {
        fillTable();
    }

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

    public ShellCommand lookup( String key )
    {
        return (ShellCommand)table.get( key );
    }

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

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

```
    }

    // Associate a command name with a ShellCommand.

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

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

    private void fillTable()
    {
        install( "newfile", new NewfileCommand() );
        install( "type", new TypeCommand() );
        install( "mkdir", new MkdirCommand() );
        install( "help", new HelpCommand() );
    }
}
```

```java
// joi/6/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 6
 */

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

    public static final String separator = File.separator;

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

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

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

```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( User owner )
    {
        this.owner = owner;
    }

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

    public User 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();
    }
}
```

```java
// joi/6/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 6
 */

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, User 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/6/jfiles/TextFile.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * A TextFile is a JFile that holds text.
 *
 * @version 6
 */

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

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

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

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

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

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

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

    /**
```

```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/6/juno/User.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * Model a Juno user.  Each User has a login name,
 * a home directory, and a real name.
 *
 * @version 6
 */

public class User
{
    private String name;     // the User’s login name
    private Directory home;  // her home Directory
    private String realName; // her real name

    /**
     * Construct a new User.
     *
     * @param name     the User’s login name.
     * @param home     her home Directory.
     * @param realName her real name.
     */

    public User( String name, Directory home, String realName )
    {
        this.name     = name;
        this.home     = home;
        this.realName = realName;
    }

    /**
     * Get the User’s login name.
     *
     * @return the name.
     */

    public String getName()
    {
        return name;
    }

    /**
     * Convert the User to a String.
     * The String representation for a User is her
     * login name.
     *
     * @return the User’s name.
     */

    public String toString()
    {
        return getName();
    }
```

```java

    /**
     * Get the User’s home Directory.
     *
     * @return the home Directory.
     */

    public Directory getHome()
    {
        return home;
    }

    /**
     * Get the user’s real name.
     *
     * @return the real name.
     */

    public String getRealName()
    {
        return realName;
    }
}
```

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

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 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, 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.
     */

```

```java
    public Bank( String bankName, Terminal atm )
    {
        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 );
        } // end of try block
        catch (NegativeAmountException e) {
            atm.errPrintln(
              "can’t start with a negative balance");
        }
        catch (InsufficientFundsException e) {
            atm.errPrintln("Initial deposit less than 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 );
                }
                else if ( transaction.startsWith( "d" ) ) {
                    int amount = readPosAmt( "    amount: " );
                    atm.println("    deposited "
                                + acct.deposit( amount ));
```

```java
                }
                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), 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 );
```

```java
        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
     */

    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();
    }

    /**
     * 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 = "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();
    }
}
```

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

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

```java

    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()
        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.
     * Does NOT count as a transaction.
     *
     * @exception InsufficientFundsException when appropriate.
```

```java
     */

    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/7/bank/CheckingAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A CheckingAccount is a BankAccount with one new feature:
 * the ability to cash a check by calling the honorCheck method.
 * Each honored check costs the customer a checkFee.
 *
 * @see BankAccount
 *
 * @version 7
 */

public class CheckingAccount extends BankAccount
{
    /**
     * Constructs a CheckingAccount with the given
     * initial balance and issuing Bank.
     * Counts as this account’s first transaction.
     *
     * @param initialBalance the opening balance for this account.
     * @param issuingBank the bank that issued this account.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public CheckingAccount( int initialBalance, Bank issuingBank )
        throws InsufficientFundsException
    {
        super( initialBalance, issuingBank );
    }

    /**
     * Honor a check:
     * Charge the account the appropriate fee
     * and withdraw the amount.
     *
     * @param amount amount (in whole dollars) to be withdrawn.
     * @return the amount withdrawn.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public int honorCheck( int amount )
        throws InsufficientFundsException
    {
        // careful error checking logic:
        // first try to deduct the check fee
        // if you succeed, try to honor check
        // if that fails, remember to add back the check fee!

        try {
            incrementBalance( - getIssuingBank().getCheckFee() );
        }
```

```java
        catch (InsufficientFundsException e) {
            throw new InsufficientFundsException(
                              "to cover check fee" );
        }
        try {
            withdraw( amount );
        }
        catch (InsufficientFundsException e) {
            incrementBalance( getIssuingBank().getCheckFee() );
            throw new InsufficientFundsException(
                              "to cover check + check fee" );
        }
        return amount;
    }

    /**
     *  Nothing special happens to a CheckingAccount on the
     *  first day of the month.
     */

    public void newMonth()
    {
        return;
    }
}
```

```java
// joi/7/bank/SavingsAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A SavingsAccount is a BankAccount that bears interest.
 * A fee is charged for too many transactions in a month.
 *
 * @see BankAccount
 *
 * @version 7
 */

public class SavingsAccount extends BankAccount
{
    private int transactionsThisMonth;

    /**
     * Override getTransactionFee() to return a non-zero fee
     * after the appropriate number of free monthly transactions.
     *
     * @return the fee for current transaction.
     */

    protected int getTransactionFee()
    {
        if (transactionsThisMonth >
            getIssuingBank().getMaxFreeTransactions()) {
            return getIssuingBank().getTransactionFee();
        }
        else {
            return 0;
        }
    }

    /**
     * Increment count of transactions, for this account for
     * this Month and in total  and for the issuing Bank, by one.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public void countTransaction()
        throws InsufficientFundsException
    {
        transactionsThisMonth++;
        super.countTransaction();
    }

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

```java
     * @exception InsufficientFundsException when appropriate.
     */

    public SavingsAccount( int initialBalance, Bank issuingBank )
         throws InsufficientFundsException
    {
        super( initialBalance, issuingBank);
        transactionsThisMonth = 1;
    }

    /**
     * A SavingsAccount earns interest each month.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public void newMonth()
         throws InsufficientFundsException
    {
        double monthlyRate = getIssuingBank().getInterestRate()/12;
        incrementBalance( (int)(monthlyRate * getBalance()));
        transactionsThisMonth = 0;
    }
}
```

```java
// joi/7/bank/FeeAccount.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A FeeAccount is a BankAccount with one new feature:
 * the user is charged for each transaction.
 *
 * @see BankAccount
 *
 * @version 7
 */

public class FeeAccount extends BankAccount
{
    /**
     * Constructor, accepting an initial balance and issuing Bank.
     *
     * @param initialBalance the opening balance.
     * @param issuingBank the bank that issued this account.
     *
     * @exception InsufficientFundsException when appropriate.
     */

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

    /**
     * The Bank’s transaction fee.
     *
     * @return the fee.
     */

    protected int getTransactionFee()
    {
        return getIssuingBank().getTransactionFee();
    }

    /**
     * The way a transaction is counted for a FeeAccount: it levies
     * a transaction fee as well as counting the transaction.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public void countTransaction()
        throws InsufficientFundsException
    {
        incrementBalance( - getTransactionFee() );
        super.countTransaction();
    }

```

```java
    /**
     * A FeeAccount incurs a monthly charge.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public void newMonth()
         throws InsufficientFundsException
    {
        incrementBalance( - getIssuingBank().getMonthlyCharge());
    }
}
```

```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.
     *
     * @exception InsufficientFundsException when appropriate.
     */

    public RegularAccount( int initialBalance, Bank issuingBank )
        throws InsufficientFundsException
    {
        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/7/bank/class Month
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

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

/**
 * The Month class implements an object that keeps
 * track of the month of the year.
 *
 * @version 7
 */

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

    private int month;
    private int year;

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

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

    /**
     *  Advance to next month.
     */

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

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

    public String toString()
```

```java
    {
        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 );
    }
}
```

```java
// joi/7/bank/InsufficientFundsException.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * Thrown when there is an attempt to spend money that is not there.
 *
 * @version 7
 */

public class InsufficientFundsException extends Exception
{
    /**
     * Construct an InsufficientFundsException
     * with a String description.
     *
     * @param msg a more specific description.
     */

    public InsufficientFundsException( String msg )
    {
        super( msg );
    }

    /**
     * Construct an InsufficientFundsException
     * with no description.
     */

    public InsufficientFundsException()
    {
        this( "" );
    }
}
```

```java
// joi/7/bank/NegativeAmountException.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * Thrown when attempting to work with a negative amount.
 *
 * @version 7
 */

public class NegativeAmountException extends Exception
{
}
```

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

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

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

public class Juno
{
    private final static String OS      = "Juno";
    private final static String VERSION = "7";

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

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

    private ShellCommandTable commandTable; // shell commands

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

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

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

        // the file system

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

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

        // create, then start a command line login interpreter

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

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

    public String getHostName()
    {
        return hostName;
    }

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

    public String getOS()
    {
        return OS;
    }

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

    public String getVersion()
    {
        return VERSION;
    }

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

    public Directory getUserHomes()
    {
        return userHomes;
    }

    /**
     * The shell command table for this system.
     *
     * @return the shell command table.
     */

    public ShellCommandTable getCommandTable()
    {
        return commandTable;
    }

    /**
     * Look up a user by user name.
     *
     * @param username the user’s name.
     * @return the appropriate User object.
     */

    public User lookupUser( String username )
    {
        return (User) users.get( username );
    }

    /**
     * Create a new User.
     *
     * @param userName the User’s login name.
     * @param home her home Directory.
     * @param realName her real name.
     * @return newly created User.
     */

    public User createUser( String userName, Directory home,
                            String realName )
    {
        User newUser = new User( userName, home, realName );
        users.put( userName, newUser );
        return newUser;
    }

    /**
     * The Juno system may be given the following command line
     * arguments.
     * <pre>
     *
     * -e:         Echo all input (useful for testing).
     *
     * -version:   Report the version number and exit.
     *
     * [hostname]: The name of the host on which
     *             Juno is running (optional).
     * </pre>
     */

    public static void main( String[] args )
```

```
    {
        // Parse command line options

        boolean echoInput = false;
        String  hostName  = "mars";

        for (int i=0; i < args.length; i++) {
            if (args[i].equals("-version")) {
                System.out.println( OS + " version " + VERSION );
                System.exit(0);
            }
            if (args[i].equals("-e")) {
                echoInput = true;
            }
            else {
                hostName = args[i];
            }
        }

        // create a Juno instance, which will start itself

        new Juno( hostName, echoInput );
    }
}
```

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

import java.util.*;

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

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

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

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

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

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

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

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

    public void CLILogin()
    {
```

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

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

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

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

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

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

    private void welcome()
    {
        console.println( "Welcome to " + system.getHostName() +
                         " running " + system.getOS() +
                         " version " + system.getVersion() );
        help();
    }

    // Remind user of available commands.

    private void help()
    {
        console.println( LOGIN_COMMANDS );
        console.println("");
    }
}
```

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

import java.util.*;

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

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

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

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

    // Run the command line interpreter

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

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

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

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

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

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

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

    public String getPrompt()
    {
        return system.getHostName() + "> ";
    }

    /**
     * The User associated with this shell.
     *
     * @return the user.
     */

    public User getUser()
    {
        return user;
    }

    /**
     * The current working directory for this shell.
     *
     * @return the current working directory.
     */

    public Directory getDot()
    {
        return dot;
    }

    /**
     * Set the current working directory for this Shell.
     *
     * @param dot the new working directory.
     */

    public void setDot(Directory dot)
    {
        this.dot = dot;
    }

    /**
     * The console associated with this Shell.
     *
     * @return the console.
     */

    public Terminal getConsole()
    {
        return console;
    }

    /**
     * The Juno object associated with this Shell.
     *
     * @return the Juno instance that created this Shell.
     */

    public Juno getSystem()
    {
        return system;
```

```
    }
}
```

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

import java.util.*;

/**
 * Model those features common to all ShellCommands.
 *
 * Each concrete extension of this class provides a constructor
 * and an implementation for method doIt.
 *
 * @version 7
 */

public abstract class ShellCommand
{
    private String helpString;   // documents the command
    private String argString;    // any args to the command

    /**
     * A constructor, always called (as super()) by the subclass.
     * Used only for commands that have arguments.
     *
     * @param helpString a brief description of what the command does.
     * @param argString a prototype illustrating the required arguments.
     */

    protected ShellCommand( String helpString, String argString )
    {
        this.argString  = argString;
        this.helpString = helpString;
    }

    /**
     * A constructor for commands having no arguments.
     *
     * @param helpString a brief description of what the command does.
     */

    protected ShellCommand( String helpString )
    {
        this( helpString, "" );
    }

    /**
     * Execute the command.
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     *
     * @exception JunoException for reporting errors
     */

    public abstract void doIt( StringTokenizer args, Shell sh )
```

```java
        throws JunoException;

    /**
     * Help for this command.
     *
     * @return the help string.
     */

    public String getHelpString()
    {
        return helpString;
    }

    /**
     * The argument string prototype.
     *
     * @return the argument string prototype.
     */

    public String getArgString()
    {
        return argString;
    }
}
```

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

import java.util.*;

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

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

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

    public ShellCommandTable()
    {
        fillTable();
    }

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

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

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

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

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

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

    // Associate a command name with a ShellCommand.

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

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

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

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

import java.util.*;

/**
 * The Juno shell command to create a new directory.
 * Usage:
 * <pre>
 *     mkdir directory-name
 * </pre>
 *
 * @version 7
 */

public class MkdirCommand extends ShellCommand
{
    MkdirCommand()
    {
        super( "create a subdirectory of the current directory",
               "directory-name" );
    }

    /**
     * Create a new Directory in the current Directory.
     *
     * @param args the remainder of the command line.
     * @param sh the current shell.
     *
     * @exception JunoException for reporting errors.
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        String filename = args.nextToken();
        new Directory( filename, sh.getUser(), sh.getDot() );
    }
}
```

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

import java.util.*;

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

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

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

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

```java
        }                                                                          // EEE
    }
}
```

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

import java.util.*;

/**
 * The Juno shell command to display help on the shell commands.
 * Usage:
 * <pre>
 *     help
 * </pre>
 *
 * @version 7
 */

public class HelpCommand extends ShellCommand
{
    HelpCommand()
    {
        super( "display ShellCommands" );
    }

    /**
     * Print out help for all commands.
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        // Get command keys from global table, print them out.

        sh.getConsole().println( "shell commands" );
        ShellCommandTable table = sh.getSystem().getCommandTable();
        String[] names = table.getCommandNames();
        for (int i = 0; i < names.length; i++ ) {
            String cmdname = names[i];
            ShellCommand cmd =
                (ShellCommand) table.lookup( cmdname );
            sh.getConsole().
                println( "  " + cmdname + ": " + cmd.getHelpString() );
        }
    }
}
```

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

import java.util.*;

/**
 * The Juno shell command to create a text file.
 * Usage:
 * <pre>
 *     newfile filename contents
 * </pre>
 *
 * @version 7
 */

public class NewfileCommand extends ShellCommand
{
    NewfileCommand()
    {
        super( "create a new TextFile", "filename contents" );
    }

    /**
     * Create a new TextFile in the current Directory.
     *
     * @param args the remainder of the command line.
     * @param sh the current shell.
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        String filename;
        String contents;
        filename = args.nextToken();
        contents  = args.nextToken("").trim(); // rest of line
        new TextFile( filename, sh.getUser(),
                      sh.getDot(), contents );
    }
}
```

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

import java.util.*;

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

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

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

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

```java
        sh.setDot( d );
    }
}
```

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

import java.util.*;

/**
 * The Juno shell command to list contents of the current directory.
 * Usage:
 * <pre>
 *     list
 * </pre>
 *
 * @version 7
 */

public class ListCommand extends ShellCommand
{
    // The constructor adds this object to the global table.

    ListCommand()
    {
        super( "list contents of current directory" );
    }

    /**
     * List contents of the current working directory.
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
        throws JunoException
    {
        Terminal terminal = sh.getConsole();
        Directory dir     = sh.getDot();
        String[] fileNames = dir.getFileNames();

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

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

import java.util.*;

/**
 * The Juno shell command to log out.
 * Usage:
 * <pre>
 *     logout
 * </pre>
 *
 * @version 7
 */

public class LogoutCommand extends ShellCommand
{
    LogoutCommand()
    {
        super( "log out, return to login: prompt" );
    }

    /**
     * Log out from the current shell.
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        throw new ExitShellException();
    }
}
```

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

import java.util.*;

/**
 * The Juno shell command to remove a text file.
 * Usage:
 * <pre>
 *     remove textfile
 * </pre>
 *
 * @version 7
 */

public class RemoveCommand extends ShellCommand
{
    RemoveCommand()
    {
        super( "remove a TextFile", "textfile" );
    }

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

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        String filename = args.nextToken();
        sh.getDot().removeJFile(filename);
    }
}
```

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

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

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

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

    public static final String separator = File.separator;

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

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

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

```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( User owner )
    {
        this.owner = owner;
    }

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

    public User 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();
    }
}
```

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

import java.util.*;

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

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

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

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

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

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

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

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

```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;
    }

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

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

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

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

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

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

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

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

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

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

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

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

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

    /**
```

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

/**
 * Model a Juno user.  Each User has a login name,
 * a home directory, and a real name.
 *
 * @version 7
 */

public class User
{
    private String name;     // the User's login name
    private Directory home;  // her home Directory
    private String realName; // her real name

    /**
     * Construct a new User.
     *
     * @param name     the User's login name.
     * @param home     her home Directory.
     * @param realName her real name.
     */

    public User( String name, Directory home, String realName )
    {
        this.name     = name;
        this.home     = home;
        this.realName = realName;
    }

    /**
     * Get the User's login name.
     *
     * @return the name.
     */

    public String getName()
    {
        return name;
    }

    /**
     * Convert the User to a String.
     * The String representation for a User is her
     * login name.
     *
     * @return the User's name.
     */

    public String toString()
    {
        return getName();
    }
```

```java

    /**
     * Get the User's home Directory.
     *
     * @return the home Directory.
     */

    public Directory getHome()
    {
        return home;
    }

    /**
     * Get the user's real name.
     *
     * @return the real name.
     */

    public String getRealName()
    {
        return realName;
    }
}
```

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

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

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

    public JunoException()
    {
    }

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

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

        super( message );
    }

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

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

/**
 * The Exception generated when a ShellCommand is misused.
 *
 * @version 7
 */

class BadShellCommandException extends JunoException
{
    private ShellCommand command;

    /**
     * Construct a new BadShellCommandException
     * containing the badly used command.
     *
     * @param the ShellCommand being misused.
     */

    public BadShellCommandException( ShellCommand command )
    {
        this.command = command;
    }

    /**
     * Get the command.
     */

    public ShellCommand getCommand()
    {
        return command;
    }
}
```

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

/**
 * Exception raised for exiting a shell.
 *
 * @version 7
 */

public class ExitShellException extends JunoException
{
}
```

```java
// joi/8/terminal/Terminal.java
// (and terminal/Terminal.java)
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.io.*;

/**
 * Terminal provides a user-friendly interface to the standard System
 * input and output streams (in, out, and err).
 * <p>
 * A Terminal is an object.  In general, one is expected to instantiate
 * just one Terminal.  Although one might instantiate several, all will
 * share the same System streams.
 * <p>
 * A Terminal may either explicitly echo input, or not.  Echoing input
 * is useful, for example, when testing with I/O redirection.
 * <p>
 * Inspired by Cay Horstmann’s Console Class.
 */

public class Terminal
{
    private boolean echo = false;
    private static BufferedReader in =
        new BufferedReader(new FileReader(FileDescriptor.in));


    // Print a prompt to the console without a newline.

    private void printPrompt( String prompt )
    {
        print( prompt );
        System.out.flush();
    }

    /**
     * Construct a Terminal that doesn’t echo input.
     */

    public Terminal()
    {
        this( false );
    }

    /**
     * Construct a Terminal.
     *
     * @param echo whether or not input should be echoed.
     */

    public Terminal( boolean echo )
    {
        this.echo = echo;
    }

```

```java
    /**
     * Read a line (terminated by a newline) from the Terminal.
     * @param prompt output string to prompt for input.
     *
     * @return the string (without the newline character),
     *  null if eof.
     */

    public String readLine( String prompt )
    {
        printPrompt(prompt);
        try {
            String line = in.readLine();
            if (echo) {
                println(line);
            }
            return line;
        }
        catch (IOException e) {
            return null;
        }
    }

    /**
     * Read a line (terminated by a newline) from the Terminal.
     *
     * @return the string (without the newline character).
     */

    public String readLine()
    {
        return readLine( "" );
    }

    // Read a line from the Terminal.  An end of file,
    // indicated by a null, raises a runtime exception.
    // Used only internally.

    private String readNonNullLine()
    {
        return readNonNullLine( "" );
    }

    // Read a line from the Terminal.  An end of file,
    // indicated by a null, raises a runtime exception.
    // Used only internally.

    private String readNonNullLine( String prompt )
    {
        String line = readLine( prompt );
        if (line == null ) {
            throw new RuntimeException( "End of File encountered." );
        }
        return line;
    }

    /**
     * Read a word from the Terminal.
     * If an empty line is entered, try again.
     * Words are terminated by whitespace.
     * Leading whitespace is trimmed; the rest of the line
     * is disposed of.
     *
     * @param prompt output string to prompt for input.
     *
     * @return the word read.
     */

    public String readWord( String prompt )
    {
        String line = readNonNullLine( prompt );
        if (line.length() == 0) {
            println( "Empty line.  Please try again." );
            return readWord("");
        }
        line = line.trim();
        for ( int i = 0; i < line.length(); i++ ) {
            if ( Character.isWhitespace( line.charAt(i) ) ) {
                return line.substring( 0, i );
            }
        }
        return line;
    }

    /**
     * Read a word from the Terminal.
     * If an empty line is entered, try again.
     * Words are terminated by whitespace.
     * Leading whitespace is trimmed; the rest of the line
     * is disposed of.
     *
     * @return the word read.
     */

    public String readWord()
    {
        return readWord( "" );
    }

    /**
     * Read a word from the Terminal.
     * If an empty line is entered, throw an exception.
     * Words are terminated by whitespace.
     * Leading whitespace is trimmed; the rest of the line
     * is disposed of.
     *
     * @param prompt output string to prompt for input.
     * @return the word read.
     *
     * @throws RuntimeException if it reads an empty line.
     */

```

```
    public String readWordOnce( String prompt )
    {
        String line = readNonNullLine( prompt );
        if (line.length() == 0) {
            throw new RuntimeException("Empty line encountered.");
        }
        line = line.trim();
        for ( int i = 0; i < line.length(); i++ ) {
            if ( Character.isWhitespace( line.charAt(i) ) ) {
                return line.substring( 0, i );
            }
        }
        return line;
    }

    /**
     * Read a word from the Terminal.
     * If an empty line is entered, throw an exception.
     * Words are terminated by whitespace.
     * Leading whitespace is trimmed; the rest of the line
     * is disposed of.
     *
     * @return the word read.
     *
     * @throws RuntimeException if it reads an empty line.
     */

    public String readWordOnce()
    {
        return readWordOnce( "" );
    }

    /**
     * Read a character from the Terminal.
     * Prompt again when an empty line is read.
     *
     * @param prompt output string to prompt for input.
     *
     * @return the character read.
     */

    public char readChar( String prompt )
    {
        String line = readNonNullLine(prompt);
        if (line.length() == 0) {
            println( "No character on line.  Please try again." );
            return readChar("");
        }
        return line.charAt(0);
    }

    /**
     * Read a character from the Terminal.
     * Throw an exception if an empty line is read.
     *
     * @param prompt output string to prompt for input.
     * @return the character read.
     *
     * @throws RuntimeException if it reads an empty line.
     */

    public char readCharOnce( String prompt )
    {
        String line = readNonNullLine(prompt);
        if (line.length() == 0) {
            throw new RuntimeException("Empty line encountered.");
        }
        return line.charAt(0);
    }

    /**
     * Read a character from the Terminal.
     * Prompt again when an empty line is read.
     *
     *
     * @return the character read.
     */

    public char readChar()
    {
        return readChar("");
    }

    /**
     * Read a character from the Terminal.
     * Throw an exception if an empty line is read.
     *
     * @return the character read.
     *
     * @throws RuntimeException if it reads an empty line.
     */

    public char readCharOnce()
    {
        return readCharOnce("");
    }

    /**
     * Read "yes" or "no" from the Terminal.
     * If an empty line or improper character is read,
     * try again.
     * Look only at first character and accept any case.
     *
     * @param prompt output string to prompt for input.
     * @return true if yes, false if no.
     */

    public boolean readYesOrNo( String prompt )
    {
        printPrompt( prompt );
        while ( true ) {
```

```
            char answer = readChar( " (y or n): ");
            if ( answer == 'y' || answer == 'Y' ) {
                return true;
            }
            else if ( answer == 'n' || answer == 'N' ) {
                return false;
            }
            else {
                printPrompt( "oops!" );
            }
        }
    }

    /**
     * Read "yes" or "no" from the Terminal.
     * If an empty line or improper character is read,
     * throw an exception.
     * Look only at first character and accept any case.
     *
     * @param prompt output string to prompt for input.
     * @return true if yes, false if no.
     *
     * @throws RuntimeException on improper input.
     */

    public boolean readYesOrNoOnce( String prompt )
    {
        printPrompt( prompt );
        while ( true ) {
            char answer = readCharOnce( " (y or n): ");
            if ( answer == 'y' || answer == 'Y' ) {
                return true;
            }
            else if ( answer == 'n' || answer == 'N' ) {
                return false;
            }
            else {
                throw new RuntimeException( "Must be y or n." );
            }
        }
    }

    /**
     * Read "yes" or "no" from the Terminal.
     * If an empty line or improper character is read,
     * try again. No prompting is done.
     * Look only at first character and accept any case.
     *
     * @return true if yes, false if no.
     */

    public boolean readYesOrNo()
    {
        while ( true ) {
            char answer = readChar();
            if ( answer == 'y' || answer == 'Y' ) {
```

```java
                return true;
            }
            else if ( answer == 'n' || answer == 'N' ) {
                return false;
            }
        }
    }

    /**
     * Read "yes" or "no" from the Terminal.
     * If an empty line or improper character is read,
     * throw an exception.
     * Look only at first character and accept any case.
     *
     * @return true if yes, false if no.
     *
     * @throws RuntimeException on improper input.
     */

    public boolean readYesOrNoOnce()
    {
        char answer = readCharOnce( " (y or n): ");
        if ( answer == 'y' || answer == 'Y' ) {
            return true;
        }
        else if ( answer == 'n' || answer == 'N' ) {
            return false;
        }
        else {
            throw new RuntimeException( "Must be y or n." );
        }
    }

    /**
     * Read an integer, terminated by a new line, from the Terminal.
     * If a NumberFormatException is encountered, try again.
     *
     * @param prompt output string to prompt for input.
     * @return the input value as an int.
     */

    public int readInt( String prompt )
    {
        while( true ) {
            try {
                return Integer.
                    parseInt(readNonNullLine( prompt ).trim());
            }
            catch (NumberFormatException e) {
                println( "Not an integer.  Please try again." );
            }
        }
    }

    /**
     * Read an integer, terminated by a new line, from the Terminal.
```

```java
     *
     * @param prompt output string to prompt for input.
     * @return the input value as an int.
     *
     * @throws NumberFormatException for a badly formed integer.
     */

    public int readIntOnce( String prompt )
        throws NumberFormatException
    {
        return Integer.parseInt(readNonNullLine( prompt ).trim());
    }

    /**
     * Read an integer, terminated by a new line, from the Terminal.
     * If a NumberFormatException is encountered, try again.
     *
     * @return the input value as an int.
     */

    public int readInt()
    {
        return readInt("");
    }

    /**
     * Read an integer, terminated by a new line, from the Terminal.
     *
     * @return the input value as an int.
     *
     * @throws NumberFormatException for a badly formed integer.
     */

    public int readIntOnce()
        throws NumberFormatException
    {
        return readIntOnce("");
    }

    /**
     * Read a double-precision floating point number,
     * terminated by a newline, from the Terminal.
     * If a NumberFormatException is encountered, try again.
     *
     * @param prompt output string to prompt for input.
     * @return the input value as a double.
     */

    public double readDouble( String prompt )
    {
        while( true ) {
            try {
                return Double.
                    parseDouble(readNonNullLine( prompt ).trim());
            }
            catch (NumberFormatException e) {
                println("Not a floating point number. Please try again.");
              }
          }
      }

      /**
       * Read a double-precision floating point number,
       * terminated by a newline, from the Terminal.
       *
       * @param prompt output string to prompt for input.
       * @return the input value as a double.
       *
       * @throws NumberFormatException for a badly formed number.
       */

      public double readDoubleOnce( String prompt )
          throws NumberFormatException
      {
          return Double.parseDouble(readNonNullLine( prompt ).trim());
      }

      /**
       * Read a double-precision floating point number,
       * terminated by a newline, from the Terminal.
       * If a NumberFormatException is encountered, try again.
       *
       * @return the input value as a double.
       */

      public double readDouble()
      {
          return readDouble("");
      }

      /**
       * Read a double-precision floating point number,
       * terminated by a newline, from the Terminal.
       *
       * @return the input value as a double.
       *
       * @throws NumberFormatException for a badly formed number.
       */

      public double readDoubleOnce()
          throws NumberFormatException
      {
          return readDouble("");
      }

      /**
       * Print a Boolean value
       * (<code>true</code> or <code>false</code>)
       * to standard output (without a newline).
       *
       * @param b Boolean to print.
       */
```

```java

      public void print( boolean b )
      {
          System.out.print( b );
      }

      /**
       * Print character to standard output (without a newline).
       *
       * @param ch character to print.
       */

      public void print( char ch )
      {
          System.out.print( ch );
      }

      /**
       * Print character array to standard output (without a newline).
       *
       * @param s character array to print.
       */

      public void print( char[] s )
      {
          System.out.print( s );
      }

      /**
       * Print a double-precision floating point number to standard
       * output (without a newline).
       *
       * @param val number to print.
       */

      public void print( double val )
      {
          System.out.print( val );
      }

      /**
       * Print a floating point number to standard output
       * (without a newline).
       *
       * @param val number to print.
       */

      public void print( float val )
      {
          System.out.print( val );
      }

      /**
       * Print integer to standard output (without a newline).
       *
       * @param val integer to print.
```

```java
     */

    public void print( int val )
    {
        System.out.print( val );
    }

    /**
     * Print a long integer to standard output (without a newline).
     *
     * @param val integer to print.
     */

    public void print( long val )
    {
        System.out.print( val );
    }

    /**
     * Print Object to standard output (without a newline).
     *
     * @param val Object to print.
     */

    public void print( Object val )
    {
        System.out.print( val.toString() );
    }

    /**
     * Print string to standard output (without a newline).
     *
     * @param str String to print.
     */

    public void print( String str )
    {
        System.out.print( str );
    }

    /**
     * Print a newline to standard output,
     * terminating the current line.
     */

    public void println()
    {
        System.out.println();
    }

    /**
     * Print a Boolean value
     * (<code>true</code> or <code>false</code>)
     * to standard output, followed by a newline.
     * @param b Boolean to print.
     */
```

```java

    public void println( boolean b )
    {
        System.out.println( b );
    }

    /**
     * Print character to standard output, followed by a newline.
     *
     * @param ch character to print.
     */

    public void println( char ch )
    {
        System.out.println( ch );
    }

    /**
     * Print a character array to standard output,
     * followed by a newline.
     *
     * @param s character array to print.
     */

    public void println( char[] s )
    {
        System.out.println( s );
    }

    /**
     * Print floating point number to standard output,
     * followed by a newline.
     *
     * @param val number to print.
     */

    public void println( float val )
    {
        System.out.println( val );
    }

    /**
     * Print a double-precision floating point number to standard
     * output, followed by a newline.
     *
     * @param val number to print.
     */

    public void println( double val )
    {
        System.out.println( val );
    }

    /**
     * Print integer to standard output, followed by a newline.
     *
     * @param val integer to print.
     */

    public void println( int val )
    {
        System.out.println( val );
    }

    /**
     * Print a long integer to standard output,
     * followed by a newline.
     *
     * @param val long integer to print.
     */

    public void println( long val )
    {
        System.out.println( val );
    }

    /**
     * Print Object to standard output, followed by a newline.
     *
     * @param val Object to print
     */

    public void println( Object val )
    {
        System.out.println( val.toString() );
    }

    /**
     * Print string to standard output, followed by a newline.
     *
     * @param str String to print
     */

    public void println( String str )
    {
        System.out.println( str );
    }

    /**
     * Print a Boolean value
     * (<code>true</code> or <code>false</code>)
     * to standard err (without a newline).
     *
     * @param b Boolean to print.
     */

    public void errPrint( boolean b )
    {
        System.err.print( b );
    }

    /**
```

```
     * Print character to standard err (without a newline).
     *
     * @param ch character to print.
     */

    public void errPrint( char ch )
    {
        System.err.print( ch );
    }

    /**
     * Print character array to standard err (without a newline).
     *
     * @param s character array to print.
     */

    public void errPrint( char[] s )
    {
        System.err.print( s );
    }

    /**
     * Print a double-precision floating point number to standard
     * err (without a newline).
     *
     * @param val number to print.
     */

    public void errPrint( double val )
    {
        System.err.print( val );
    }

    /**
     * Print a floating point number to standard err
     * (without a newline).
     *
     * @param val number to print.
     */

    public void errPrint( float val )
    {
        System.err.print( val );
    }

    /**
     * Print integer to standard err (without a newline).
     *
     * @param val integer to print.
     */

    public void errPrint( int val )
    {
        System.err.print( val );
    }

    /**
     * Print a long integer to standard err (without a newline).
     *
     * @param val integer to print.
     */

    public void errPrint( long val )
    {
        System.err.print( val );
    }

    /**
     * Print Object to standard err (without a newline).
     *
     * @param val Object to print.
     */

    public void errPrint( Object val )
    {
        System.err.print( val.toString() );
    }

    /**
     * Print string to standard err (without a newline).
     *
     * @param str String to print.
     */

    public void errPrint( String str )
    {
        System.err.print( str );
    }

    /**
     * Print a newline to standard err,
     * terminating the current line.
     */

    public void errPrintln()
    {
        System.err.println();
    }

    /**
     * Print a Boolean value
     * (<code>true</code> or <code>false</code>)
     * to standard err, followed by a newline.
     *
     * @param b Boolean to print.
     */

    public void errPrintln( boolean b )
    {
        System.err.println( b );
    }

```

```
    /**
     * Print character to standard err, followed by a newline.
     *
     * @param ch character to print.
     */

    public void errPrintln( char ch )
    {
        System.err.println( ch );
    }

    /**
     * Print a character array to standard err,
     * followed by a newline.
     *
     * @param s character array to print.
     */

    public void errPrintln( char[] s )
    {
        System.err.println( s );
    }

    /**
     * Print floating point number to standard err,
     * followed by a newline.
     *
     * @param val number to print.
     */

    public void errPrintln( float val )
    {
        System.err.println( val );
    }

    /**
     * Print a double-precision floating point number to
     * standard err, followed by a newline.
     *
     * @param val number to print.
     */

    public void errPrintln( double val )
    {
        System.err.println( val );
    }

    /**
     * Print integer to standard err, followed by a newline.
     *
     * @param val integer to print.
     */

    public void errPrintln( int val )
    {
        System.err.println( val );
    }

    /**
     * Print a long integer to standard err, followed by a newline.
     *
     * @param val long integer to print.
     */

    public void errPrintln( long val )
    {
        System.err.println( val );
    }

    /**
     * Print Object to standard err, followed by a newline.
     *
     * @param val Object to print
     */

    public void errPrintln( Object val )
    {
        System.err.println( val.toString() );
    }

    /**
     * Print string to standard err, followed by a newline.
     *
     * @param str String to print
     */

    public void errPrintln( String str )
    {
        System.err.println( str );
    }

    /**
     * Unit test for Terminal.
     *
     * @param args command line arguments:
     * <pre>
     *    -e echo all input.
     * </pre>
     */

    public static void main( String[] args )
    {
        Terminal t =
            new Terminal( args.length == 1 && args[0].equals("-e") );

        String line = t.readLine( "line:" );
        String word = t.readWord( "word:" );
        char   c    = t.readChar( "char:" );
        boolean yn  = t.readYesOrNo( "yorn:" );
        double d    = t.readDouble( "double:" );
        int    i    = t.readInt( "int:" );

```

```
        t.print("  line:[");  t.print(line);    t.print("]");
        t.print("  line:[");  t.println(line);  t.print("]");

        t.print("  word:[");  t.print(word);    t.print("]");
        t.print("  word:[");  t.println(word);  t.print("]");

        t.print("  char:[");  t.print(c);       t.print("]");
        t.print("  char:[");  t.println(c);     t.print("]");

        t.print("  yorn:[");  t.print(yn);      t.print("]");
        t.print("  yorn:[");  t.println(yn);    t.print("]");

        t.print("  doub:[");  t.print(d);       t.print("]");
        t.print("  doub:[");  t.println(d);     t.print("]");

        t.print("  int:[");   t.print(i);       t.print("]");
        t.print("  int:[");   t.println(i);     t.print("]");

        t.errPrint("  line:[");  t.errPrint(line);    t.errPrint("]");
        t.errPrint("  line:[");  t.errPrintln(line);  t.errPrint("]");

        t.errPrint("  word:[");  t.errPrint(word);    t.errPrint("]");
        t.errPrint("  word:[");  t.errPrintln(word);  t.errPrint("]");

        t.errPrint("  char:[");  t.errPrint(c);       t.errPrint("]");
        t.errPrint("  char:[");  t.errPrintln(c);     t.errPrint("]");

        t.errPrint("  yorn:[");  t.errPrint(yn);      t.errPrint("]");
        t.errPrint("  yorn:[");  t.errPrintln(yn);    t.errPrint("]");

        t.errPrint("  doub:[");  t.errPrint(d);       t.errPrint("]");
        t.errPrint("  doub:[");  t.errPrintln(d);     t.errPrint("]");

        t.errPrint("  int:[");   t.errPrint(i);       t.errPrint("]");
        t.errPrint("  int:[");   t.errPrintln(i);     t.errPrint("]");
    }
}
```

```java
// joi/8/juno/Password.java//
//
//
//  Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * Model a good password.
 *
 * <p>
 * A password is a String satisfying the following conditions
 * (close to those required of Unix passwords, according to
 * the <code> man passwd </code> command in Unix):
 * <br>
 * <ul>
 * <li> A password must have at least PASSLENGTH characters, where
 *      PASSLENGTH defaults to 6. Only the first eight characters
 *      are significant.
 *
 * <li> A password must contain at least two alphabetic characters
 *      and at least one numeric or special character. In this  case,
 *      "alphabetic" refers to all upper or lower case letters.
 *
 * <li> A password must not contain a specified string as a substring
 *      For comparison purposes, an upper case letter and its
 *      corresponding lower case letter are equivalent.
 *
 * <li> A password must not be a substring of a specified string.
 *      For comparison purposes, an upper case letter and its
 *      corresponding lower case letter are equivalent.
 *
 * </ul>
 * <br>
 * A password string may be stored in a Password object only in
 * encrypted form.
 */

public class Password
{
    private String password;

    /**
     * Construct a new Password.
     *
     * @param password the new password.
     * @param notSubstringOf a String that may not contain the password.
     * @param doesNotContain a String the password may not contain.
     *
     * @exception BadPasswordException when password is unacceptable.
     */

    public Password(String password, String notSubstringOf,
                    String doesNotContain)
         throws BadPasswordException
    {
        // if password is not acceptable
        // throw new BadPasswordException( reason )
```

```java
        this.password = encrypt(password);
    }

    // Rewrite s in a form that makes it hard to guess s.

    private String encrypt( String s )
    {
        return Integer.toHexString(s.hashCode());
    }

    /**
     * See whether a supplied guess matches this password.
     *
     * @param guess the trial password.
     *
     * @exception BadPasswordException when match fails.
     */

    public void match(String guess)
         throws BadPasswordException
    {
    }

    /**
     * Unit test for Password objects.
     */

    public static void main( String[] args )
    {
    }
}
```

```java
// joi/8/juno/BadPasswordException.java
//
//
//  Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * The exception thrown when an initial password is unacceptable
 * or a match against an existing password fails.
 */

public class BadPasswordException extends Exception
{
    BadPasswordException()
    {
        super();
    }

    BadPasswordException(String message)
    {
        super(message);
    }
}
```

```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.");
            }
        }
    }
}
```

```
// 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.
```

```
     *
     * @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 );
    }

```

```java
    // 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
     */

```

```
    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(
```

```
                               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 );
    }
}
```

```java
// joi/10/joi/JOIPanel.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/**
 * A JOIPanel displays a button and a message.
 * Pushing the button changes the message.
 *
 * This panel can be displayed either from an applet
 * in a browser or by the JVM as an application.
 *
 * @version 10
 */

public class JOIPanel extends Applet
{
    private static final String MESSAGE1 = "Java Outside In";
    private static final String MESSAGE2 = "Java Inside Out";
    private String currentMessage = MESSAGE1; // currently displayed

    private Font font;                // for printing the message
    private Button button;            // for changing messages

    /**
     * Equip this Panel with a Button
     * and an associated ButtonListener, and
     * set the font for the message.
     */

    public void init()
    {
        // what this Panel looks like
        button = new Button( "Press Me" );
        this.add( button );
        font = new Font("Garamond", Font.BOLD, 48);

        // how this Panel behaves
        button.addActionListener( new JOIButtonListener( this ) );
    }

    /**
     * Method that responds when the ButtonListener sends a
     * changeMessage message.
     */

    public void changeMessage()
    {
        currentMessage =
            currentMessage.equals(MESSAGE1) ? MESSAGE2 : MESSAGE1;
        this.repaint();
    }
```

```java

    /**
     * Draw the current message on this Panel.
     *
     * (The button is already there.)
     *
     * @param g an object encapsulating the graphics (e.g. pen)
     *          properties.
     */

    public void paint(Graphics g)
    {
        g.setColor(Color.black);
        g.setFont(font);
        g.drawString(currentMessage, 40, 75);
    }

    /**
     * Ask the JVM to display this Panel.
     */

    public static void main( String[] args )
    {
        Terminal t     = new Terminal();
        Frame frame    = new Frame();
        JOIPanel panel = new JOIPanel();
        panel.init();
        frame.add(panel);
        frame.setSize(400,120);
        frame.show();
        t.readLine("Type return to close the window ... ");
        System.exit(0);
    }
}
```

```java
// joi/10/joi/JOIButtonListener.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.awt.event.*;

/**
 * A simple listener for responding to button presses.
 * It knows the Panel on which the button lives, and
 * responds to button events by sending a changeMessage()
 * to that Panel.
 *
 * @version 10
 */

public class JOIButtonListener implements ActionListener
{
    private JOIPanel panel;  // the Panel containing the Button

    /**
     * Construct the ButtonListener.
     *
     * @param panel the Panel on which this Button will act.
     */

    public JOIButtonListener( JOIPanel panel )
    {
        this.panel = panel;
    }

    /**
     * Defines the ActionListener behavior that must be implemented.
     *
     * When a user pushes the Button that we’re listening to,
     * send a changeMessage() message to the Panel.
     *
     * @param e the "event" when the button is pressed.
     */

    public void actionPerformed( ActionEvent e )
    {
        panel.changeMessage();
    }
}
```

```
<!-- joi/10/joi/joi.html-->
<!-- -->
<!-- -->
<!-- Copyright 2002 Bill Campbell and Ethan Bolker-->

<html>
<body>

<applet
code="JOIPanel.class" height=100 width=400>
</applet>

</html>
</body>
```

```java
// joi/10/joiapplet/JOIApplet.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.applet.*;
import java.awt.*;
import java.awt.event.*;

/**
 * A JOIPanel displays a button and a message.
 * Pushing the button changes the message.
 *
 * This class provides both the panel and the listener for
 * the button on the panel - a common GUI programming idiom.
 *
 * The panel can be displayed either from an applet
 * in a browser or by the JVM as an application.
 *
 * @version 10
 *
 */

public class JOIApplet extends Applet implements ActionListener
{
    private static final String MESSAGE1 = "Java Outside In";
    private static final String MESSAGE2 = "Java Inside Out";
    private String currentMessage = MESSAGE1; // currently displayed

    private Font font;                // for printing the message
    private Button button;            // for changing messages

    /**
     * Equip this Panel with a Button
     * and an associated ButtonListener, and
     * set the font for the message.
     */

    public void init()
    {
        // what this Panel looks like
        button = new Button( "Press Me" );
        this.add( button );
        font = new Font("Garamond", Font.BOLD, 48);

        // how this Panel behaves
        button.addActionListener( this );
    }

    /**
     * Defines the ActionListener behavior that must be
     * implemented.
     *
     * When a user pushes the Button that we're listening to,
     * send a changeMessage() message to the Panel.
     *
```

```java
     * @param e the "event" when the button is pressed.
     */

    public void actionPerformed( ActionEvent e )
    {
        currentMessage =
            currentMessage.equals(MESSAGE1) ? MESSAGE2 : MESSAGE1;
        this.repaint();
    }

    /**
     * Draw the current message on this Panel.
     *
     * (The button is already there.)
     *
     * @param g an object encapsulating the graphics (e.g. pen)
     *          properties.
     */

    public void paint(Graphics g)
    {
        g.setColor(Color.black);
        g.setFont(font);
        g.drawString(currentMessage, 40, 75);
    }

    /**
     * Ask the JVM to display this Panel.
     */

    public static void main( String[] args )
    {
        Terminal t     = new Terminal();
        Frame frame    = new Frame();
        JOIApplet panel = new JOIApplet();
        panel.init();
        frame.add(panel);
        frame.setSize(400,120);
        frame.show();
        t.readLine("Type return to close the window ... ");
        System.exit(0);
    }
}
```

```
<!-- joi/10/joiapplet/classes/joiapplet.html-->
<!-- -->
<!-- -->
<!-- Copyright 2002 Bill Campbell and Ethan Bolker-->

<html>
<body>

<applet
code="JOIApplet.class" height=100 width=400>
</applet>

</html>
</body>
```

```java
// joi/10/juno/Juno.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

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

/**
 * Juno (Juno’s Unix NOt) mimics a command line operating system
 * such as Unix.
 * <p>
 * A Juno system has a name, a set of Users, a JFile system,
 * a login process and a set of shell commands.
 *
 * @see User
 * @see JFile
 * @see ShellCommand
 *
 * @version 10
 **/

public class Juno
    implements Serializable
{
    private final static String OS      = "Juno";
    private final static String VERSION = "10";

    private String    hostName;   // host machine name
    private Map       users;      // lookup table for Users
    private transient OutputInterface  console;

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

    private ShellCommandTable commandTable; // shell commands

    // file containing Juno state
    private transient String fileName = null;

    // port used by Juno server for remote login
    private int junoPort = 2001;

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

    public Juno( String hostName, boolean echoInput,
                 boolean isGUI, boolean isRemote )
    {
```

```java
        // Initialize the Juno environment ...

        this.hostName     = hostName;
        users             = new TreeMap();
        commandTable      = new ShellCommandTable();

        // the file system

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

    // Set up the correct console:
    // command line (default), graphical or remote.

    private void setupConsole( boolean echoInput, boolean isGUI,
                               boolean isRemote )
    {
        LoginInterpreter interpreter
            = new LoginInterpreter( this, null );

        if (isGUI) {
            console = new GUILoginConsole( hostName,
                                           this, interpreter, echoInput);
        }
        else if (isRemote) {
            console = new RemoteConsole( this, echoInput, junoPort );
        }
        else {
            console = new JunoTerminal( echoInput );
        }

        // Tell the interpreter about the console
        interpreter.setConsole( console );

        // If we’re using a simple command line interface,
        // start that.  (Constructing a GUI starts the GUI.)
        // Shut down Juno when done

        if (!isGUI && !isRemote) {
            interpreter.CLILogin();
            shutDown();
        }
    }

    /**
     * Shut down this Juno system.
     *
     * Save state if required.
     */

    public void shutDown( )
    {
        if (fileName != null) {
            writeJuno( );
        }
    }

    /**
     * Set the name of file in which system state is kept.
     *
     * @param fileName the file name.
     */

    public void setFileName(String fileName)
    {
        this.fileName = fileName;
    }

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

    public String getHostName()
    {
        return hostName;
    }

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

    public String getOS()
    {
        return OS;
    }

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

    public String getVersion()
    {
        return VERSION;
    }

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

```java

    public Directory getUserHomes()
    {
        return userHomes;
    }

    /**
     * The shell command table for this system.
     *
     * @return the shell command table.
     */

    public ShellCommandTable getCommandTable()
    {
        return commandTable;
    }

    /**
     * Look up a user by user name.
     *
     * @param username the user’s name.
     * @return the appropriate User object.
     */

    public User lookupUser( String username )
    {
        return (User) users.get( username );
    }

    /**
     * Create a new User.
     *
     * @param userName the User’s login name.
     * @param home her home Directory.
     * @param password her password.
     * @param realName her real name.
     * @return newly created User.
     */

    public User createUser( String userName, Directory home,
                            String password, String realName )
    {
        User newUser = new User( userName, password,
                                 home, realName );
        users.put( userName, newUser );
        return newUser;
    }

    /**
     * The Juno system may be given the following command line
     * arguments:
     *
     * -e:         Echo all input (useful for testing).
     *
     * -version:   Report the version number and exit.
     *
     * -g:         Support a GUI console.
     *
     * -remote     Start Juno server.
     *
     * -f filename File to read/write system state from/to
     *
     * [hostname]: The name of the host on which
     *             Juno is running (optional).
     */

    public static void main( String[] args )
    {
        // Parse command line options

        boolean echoInput    = false;
        boolean versionQuery = false;
        boolean isGUI        = false;
        boolean isRemote     = false;
        String  hostName     = "mars";
        String  junoFileName = null;

        for (int i=0; i < args.length; i++) {
            if (args[i].equals("-e")) {
                echoInput = true;
            }
            else if (args[i].equals("-version")) {
                versionQuery = true;
            }
            else if (args[i].equals("-g")) {
                isGUI = true;
            }
            else if (args[i].equals( "-remote" )) {
                isRemote = true;
            }
            else if (args[i].equals("-f")) {
                junoFileName = args[++i];
            }
            else {
                hostName = args[i];
            }
        }

        // If it’s  a version query give the version and exit
        if ( versionQuery ) {
            System.out.println( OS + " version " + VERSION );
            System.exit(0);
        }

        // Create a new Juno or read one from a file.
        Juno junoSystem = null;
        if (junoFileName != null) {
            junoSystem = readJuno( junoFileName );
        }
        if (junoSystem == null) {
            junoSystem = new Juno( hostName, echoInput,
                                   isGUI, isRemote );
```

```java
        }
        junoSystem.setFileName( junoFileName );
        junoSystem.setupConsole( echoInput, isGUI, isRemote );
    }

    // Read Juno state from a file.
    //
    // @param junoFileName the name of the file containing the system.
    // @return the system, null if file does not exist.

    private static Juno readJuno(String junoFileName)
    {
        File file = new File( junoFileName );
        if (!file.exists()) {
            return null;
        }
        ObjectInputStream inStream = null;
        try {
            inStream = new ObjectInputStream(
                               new FileInputStream( file ) );
            Juno juno = (Juno)inStream.readObject();
            System.out.println(
               "Juno state read from file " + junoFileName);
            return juno;
        }
        catch (Exception e ) {
            System.err.println("Problem reading " + junoFileName );
            System.err.println(e);
            System.exit(1);
        }
        finally {
            try {
                inStream.close();
            }
            catch (Exception e) {
            }
        }
        return null; // you can never get here
    }

    // Write Juno state to a file.

    private void writeJuno()
    {
        ObjectOutputStream outStream = null;
        try {
            outStream = new ObjectOutputStream(
                               new FileOutputStream( fileName ) );
            outStream.writeObject( this );
            System.out.println(
               "Juno state written to file " + fileName);
        }
        catch (Exception e ) {
            System.err.println("Problem writing " + fileName);
            System.err.println(e);
        }
        finally {
            try {
                outStream.close();
            }
            catch (Exception e ) {
            }
        }
    }
}
```

```java
// joi/10/juno/LoginInterpreter.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Interpreter for Juno login commands.
 *
 * There are so few commands that if-then-else logic is OK.
 *
 * @version 10
 */

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

    private Juno system;             // the Juno object
    private OutputInterface console; // where output goes

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

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

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

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

    /**
     * Simulates behavior at login: prompt.
     */

    public void CLILogin()
    {
```

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


    /**
     * Parse user’s command line and dispatch appropriate
     * semantic action.
     *
     * @param  inputLine the User’s instructions.
     * @return true except for "exit" command
     *         or null inputLine.
     */

    public boolean interpret( String inputLine )
    {
        if (inputLine == null) {
            return false;
        }
        StringTokenizer st =
            new StringTokenizer( inputLine );
        if (st.countTokens() == 0) {
            return true; // skip blank line
        }
        String visitor = st.nextToken();
        if (visitor.equals( "exit" )) {
            return false;
        }
        if (visitor.equals( "register" )) {
            register( st );
        }
        else if (visitor.equals( "help" )) {
            help();
        }
        else {
            String password;
            try {
                if (console.isGUI()) {
                    password = st.nextToken();
                }
                else {;
                    password = readPassword( "password: " );
                }
                User user = system.lookupUser(visitor);
                user.matchPassword( password );
                new Shell( system, user, console );
            }
            catch (Exception e) {

                // NullPointerException if no such user,
                // JunoException if password fails to match -
                // message to user doesn’t give away which.
                // The sysadmin would probably want a log
                // that did keep track.
                //
                // Other exceptions should be caught in shell()

                console.println("sorry");
            }
        }
        return true;
    }

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

    private void register( StringTokenizer line )
    {
        String username = "";
        String password = "";
        String realname = "";
        try {
            username = line.nextToken();
            password = line.nextToken();
            realname = line.nextToken("").trim();
        }
        catch (NoSuchElementException e) {
        }

        if (username.equals("") || password.equals("")
            || realname.equals("") ) {
            console.println(
                  "please supply username, password, real name");
            return;
        }
        User user = system.lookupUser(username);

        if (user != null) {  // user already exists
            console.println("sorry");
            return;
        }
        if (badPassword( password )) {
            console.println("password too easy to guess");
            return;
        }
        Directory home = new Directory( username, null,
                                        system.getUserHomes() );

        user = system.createUser( username, home, password, realname );
        home.setOwner( user );
    }

    // test to see if password is unacceptable:
    //   fewer than 6 characters
    //   contains only alphabetic characters
```

```java

    private boolean badPassword( String pwd )
    {
        if (pwd.length() < 6) {
            return true;
        }
        int nonAlphaCount = 0;
        for (int i=0; i < pwd.length(); i++) {
            if (!Character.isLetter(pwd.charAt(i))) {
                nonAlphaCount++;
            }
        }
        return (nonAlphaCount == 0);
    }

    // Used for reading the user’s password in CLI.

    private String readPassword( String prompt )
    {
        String line =
            ((InputInterface) console).readLine( prompt );
        StringTokenizer st = new StringTokenizer( line );
        try {
            return st.nextToken();
        }
        catch ( NoSuchElementException e )  {}
        return ""; // keeps compiler happy
    }


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

    private void welcome()
    {
        console.println( "Welcome to " + system.getHostName() +
                         " running "   + system.getOS() +
                         " version "   + system.getVersion() );
        help();
    }

    // Remind user of available commands.

    private void help()
    {
        console.println( LOGIN_COMMANDS );
        console.println("");
    }
}
```

```java
// joi/10/juno/Shell.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * Models a shell (command interpreter)
 *
 * The Shell knows the (Juno) system it’s working in,
 * the User who started it,
 * and the console to which to send output.
 *
 * It keeps track of the the current working directory (.) .
 *
 * @version 10
 */

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

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

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

        if (!console.isGUI()) {
            this.console = console;
            CLIShell();
        }
        else
            this.console =
                new GUIShellConsole("Juno shell for " + user,
                                    this, console.isEchoInput());
    }

    // Run the command line interpreter

    private void CLIShell()
    {
```

```java
        boolean moreWork = true;
        while(moreWork) {
            moreWork = interpret( ((InputInterface) console).
                                  readLine( getPrompt() ) );
        }
        console.println("goodbye");
    }

    /**
     * Interpret a String.
     *
     * Syntax
     * <pre>
     *    shellcommand command-arguments
     * </pre>
     *
     * @param inputLine the String to interpret.
     * @return true unless shell command is logout.
     */

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

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

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

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

    public String getPrompt()
    {
      return system.getHostName() + ":" + getDot().getPathName() + "> ";
    }

    /**
     * The User associated with this shell.
     *
     * @return the user.
     */

    public User getUser()
    {
        return user;
    }

    /**
     * The current working directory for this shell.
     *
     * @return the current working directory.
     */

    public Directory getDot()
    {
        return dot;
    }

    /**
     * Set the current working directory for this Shell.
     *
     * @param dot the new working directory.
     */

    public void setDot(Directory dot)
    {
        this.dot = dot;
    }


    /**
     * The console associated with this Shell.
```

```
     *
     * @return the console.
     */

    public OutputInterface getConsole()
    {
        return console;
    }

    /**
     * The Juno object associated with this Shell.
     *
     * @return the Juno instance that created this Shell.
     */

    public Juno getSystem()
    {
        return system;
    }
}
```

```java
// joi/10/juno/ShellCommand.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * Model those features common to all ShellCommands.
 *
 * Each concrete extension of this class provides a constructor
 * and an implementation for method doIt.
 *
 * @version 10
 */

public abstract class ShellCommand
    implements java.io.Serializable
{
    private String helpString;   // documents the command
    private String argString;    // any args to the command

    /**
     * A constructor, always called (as super()) by the subclass.
     * Used only for commands that have arguments.
     *
     * @param helpString a brief description of what the command does.
     * @param argString a prototype illustrating the required arguments.
     */

    protected ShellCommand( String helpString, String argString )
    {
        this.argString  = argString;
        this.helpString = helpString;
    }

    /**
     * A constructor for commands having no arguments.
     *
     * @param helpString a brief description of what the command does.
     */

    protected ShellCommand( String helpString )
    {
        this( helpString, "" );
    }

    /**
     * Execute the command.
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     *
     * @exception JunoException for reporting errors
     */

```

```java
    public abstract void doIt( StringTokenizer args, Shell sh )
        throws JunoException;

    /**
     * Help for this command.
     *
     * @return the help string.
     */

    public String getHelpString()
    {
        return helpString;
    }

    /**
     * The argument string prototype.
     *
     * @return the argument string prototype.
     */

    public String getArgString()
    {
        return argString;
    }
}
```

```java
// joi/10/juno/ShellCommandTable.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * A ShellCommandTable object maintains a dispatch table of
 * ShellCommand objects keyed by the command names used to invoke
 * them.
 *
 * To add a new shell command to the table, install it from
 * method fillTable().
 *
 * @see ShellCommand
 *
 * @version 10
 */

public class ShellCommandTable
    implements java.io.Serializable
{
    private Map table = new TreeMap();

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

    public ShellCommandTable()
    {
        fillTable();
    }

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

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

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

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

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

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

    // Associate a command name with a ShellCommand.

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

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

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

```java
// joi/10/juno/MkdirCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to create a new directory.
 * Usage:
 * <pre>
 *     mkdir directory-name
 * </pre>
 *
 * @version 10
 */

public class MkdirCommand extends ShellCommand
{
    MkdirCommand()
    {
        super( "create a subdirectory of the current directory",
               "directory-name" );
    }

    /**
     * Create a new Directory in the current Directory.
     *
     * @param args the remainder of the command line.
     * @param sh the current shell.
     *
     * @exception JunoException for reporting errors.
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        String filename = args.nextToken();
        new Directory( filename, sh.getUser(), sh.getDot() );
    }
}
```

```java
// joi/10/juno/TypeCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to display the contents of a
 * text file.
 * Usage:
 * <pre>
 *     type textfile
 * </pre>
 *
 * @version 10
 */

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

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

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

```java
        }
    }
}
```

```java
// joi/10/juno/HelpCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to display help on the shell commands.
 * Usage:
 * <pre>
 *     help
 * </pre>
 *
 * @version 10
 */

public class HelpCommand extends ShellCommand
{
    HelpCommand()
    {
        super( "display ShellCommands" );
    }

    /**
     * Print out help for all commands.
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        // Get command keys from global table, print them out.

        sh.getConsole().println( "shell commands" );
        ShellCommandTable table = sh.getSystem().getCommandTable();
        String[] names = table.getCommandNames();
        for (int i = 0; i < names.length; i++ ) {
            String cmdname = names[i];
            ShellCommand cmd =
                (ShellCommand) table.lookup( cmdname );
            sh.getConsole().
                println( "  " + cmdname + ": " + cmd.getHelpString() );
        }
    }
}
```

```java
// joi/10/juno/NewfileCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to create a text file.
 * Usage:
 * <pre>
 *     newfile filename contents
 * </pre>
 *
 * @version 10
 */

public class NewfileCommand extends ShellCommand
{
    NewfileCommand()
    {
        super( "create a new TextFile", "filename contents" );
    }

    /**
     * Create a new TextFile in the current Directory.
     *
     * @param args the remainder of the command line.
     * @param sh the current shell.
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        String filename;
        String contents;
        filename = args.nextToken();
        contents  = args.nextToken("").trim(); // rest of line
        new TextFile( filename, sh.getUser(),
                      sh.getDot(), contents );
    }
}
```

```java
// joi/10/juno/CdCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to change directory.
 * Usage:
 * <pre>
 *     cd [directory]
 * </pre>
 * for moving to the named directory.
 *
 * @version 10
 */

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

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

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

```java
}
```

```java
// joi/10/juno/ListCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to list contents of the current directory.
 * Usage:
 * <pre>
 *     list
 * </pre>
 *
 * @version 10
 */

public class ListCommand extends ShellCommand
{
    // The constructor adds this object to the global table.

    ListCommand()
    {
        super( "list contents of current directory" );
    }

    /**
     * List contents of the current working directory.
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
        throws JunoException
    {
        OutputInterface terminal = sh.getConsole();
        Directory dir            = sh.getDot();
        String[] fileNames       = dir.getFileNames();

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

```java
// joi/10/juno/GetfileCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

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

/**
 * The Juno shell command to get a text file from the underlying
 * operating system and copy it to a Juno text file.
 * Usage:
 * <pre>
 *     getfile native-filename juno-filename
 * </pre>

 * <pre>
 *
 * @version 10
 */

class GetfileCommand extends ShellCommand
{
    GetfileCommand()
    {
        super( "download a file to Juno",
               "native-filename juno-filename" );
    }

    /**
     * Use the getfile command to copy the content of a real
     * file to a Juno TextFile.
     * <p>
     * The command has the form:
     * <pre>
     * get nativeFile textfile <&>
     *
     * @param args: the reminder of the command line.
     * @param sh: the current shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        if ( sh.getConsole().isRemote() ) {
            throw( new JunoException(
                "Get not implemented for remote consoles." ) );
        }
        String src;
        String dst;
        try {
            src = args.nextToken();
            dst = args.nextToken();
        }
```

```java
        catch (NoSuchElementException e) {
            throw new BadShellCommandException( this );
        }
        BufferedReader inStream  = null;
        Writer         outStream = null;
        try {
            inStream  = new BufferedReader( new FileReader( src ) );
            outStream = new StringWriter();
            String line;

            while ((line = inStream.readLine()) != null) {
                outStream.write( line );
                outStream.write( '\n' );
            }
            new TextFile( dst, sh.getUser(),
                          sh.getDot(), outStream.toString() );
        }
        catch (IOException e) {
            throw new JunoException( "IO problem in get" );
        }
        finally {
            try {
                inStream.close();
                outStream.close();
            }
            catch (IOException e) {};
        }
    }
}
```

```java
// joi/10/juno/RemoveCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to remove a text file.
 * Usage:
 * <pre>
 *     remove textfile
 * </pre>
 *
 * @version 10
 */

public class RemoveCommand extends ShellCommand
{
    RemoveCommand()
    {
        super( "remove a TextFile", "textfile" );
    }

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

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        String filename = args.nextToken();
        sh.getDot().removeJFile(filename);
    }
}
```

```java
// joi/10/juno/LogoutCommand.java
//
//
// Copyright 2003, Bill Campbell and Ethan Bolker

import java.util.*;

/**
 * The Juno shell command to log out.
 * Usage:
 * <pre>
 *     logout
 * </pre>
 *
 * @version 10
 */

public class LogoutCommand extends ShellCommand
{
    LogoutCommand()
    {
        super( "log out, return to login: prompt" );
    }

    /**
     * Log out from the current shell.
     *
     * @param args the remainder of the command line.
     * @param sh   the current shell
     *
     * @exception JunoException for reporting errors
     */

    public void doIt( StringTokenizer args, Shell sh )
         throws JunoException
    {
        throw new ExitShellException();
    }
}
```

```java
// joi/10/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 10
 */

public abstract class JFile
    implements java.io.Serializable
{
    /**
     * The separator used in pathnames.
     */

    public static final String separator = File.separator;

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

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

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

```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( User owner )
    {
        this.owner = owner;
    }

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

    public User 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();
    }
}
```

```java
// joi/10/juno/Directory.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

import java.util.*;

/**
 * Directory of JFiles.
 *
 * A Directory is a JFile that maintains a
 * table of the JFiles it contains.
 *
 * @version 10
 */

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

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

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

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

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

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

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

```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;
    }

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

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

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

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

```java
// joi/10/juno/TextFile.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * A TextFile is a JFile that holds text.
 *
 * @version 10
 */

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

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

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

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

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

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

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

    /**
```

```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/10/juno/User.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * Model a juno user.  Each User has a login name, password,
 * a home directory, and a real name.
 * name.
 *
 * @version 10
 */

public class User
    implements java.io.Serializable
{
    private String name;     // the User’s login name
    private String password; // The user’s login password.
    private Directory home;  // her home Directory
    private String realName; // her real name

    /**
     * Construct a new User.
     *
     * @param name      the User’s login name.
     * @param password  the user’s login password.
     * @param home      her home Directory.
     * @param realName  her real name.
     */

    public User( String name, String password,
                 Directory home, String realName )
    {
        this.name     = name;
        this.password = password;
        this.home     = home;
        this.realName = realName;
    }

    /**
     *   Confirm password. Throw a JunoException on failure.
     *
     *   @param guess the string to test against the password.
     *
     *   @exception JunoException
     *      if password fails to match
     */

    public void matchPassword( String guess ) throws JunoException
    {
        if (!guess.equals( password )) {
            throw new JunoException( "bad password" );
        }
    }

    /**
```

```java
     * Get the User’s login name.
     *
     * @return the name.
     */

    public String getName()
    {
        return name;
    }

    /**
     * Convert the User to a String.
     * The String representation for a User is her
     * login name.
     *
     * @return the User’s name.
     */

    public String toString()
    {
        return getName();
    }

    /**
     * Get the User’s home Directory.
     *
     * @return the home Directory.
     */

    public Directory getHome()
    {
        return home;
    }

    /**
     * Get the user’s real name.
     *
     * @return the real name.
     */

    public String getRealName()
    {
        return realName;
    }
}
```

```java
// joi/10/juno/JunoException.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A general Juno Exception.
 *
 * @version 10
 */

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

    public JunoException()
    {
    }

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

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

        super( message );
    }

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

```java
// joi/10/juno/BadShellCommandException.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 * The Exception generated when a ShellCommand is misused.
 *
 * @version 10
 */

class BadShellCommandException extends JunoException
{
    private ShellCommand command;

    /**
     * Construct a new BadShellCommandException
     * containing the badly used command.
     *
     * @param the ShellCommand being misused.
     */

    public BadShellCommandException( ShellCommand command )
    {
        this.command = command;
    }

    /**
     * Get the command.
     */

    public ShellCommand getCommand()
    {
        return command;
    }
}
```

```java
// joi/10/juno/ExitShellException.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * Exception raised for exiting a shell.
 *
 * @version 10
 */

public class ExitShellException extends JunoException
{
}
```

```java
// joi/10/juno/ShellCommandNotFoundException.java (version 10)
//
//
// Copyright 1997-2001 Ethan Bolker and Bill Campbell

/**
 * The Exception when a ShellCommand isn’t found.
 */

class ShellCommandNotFoundException extends JunoException
{
    /**
     * Create a ShellCommandNotFoundException.
     */

    public ShellCommandNotFoundException()
    {
    }

    /**
     * Create a ShellCommandNotFoundException with
     * a message reporting the command tried.
     */

    public ShellCommandNotFoundException(String commandName )
    {
        super( "ShellCommand " + commandName + " not found" );
    }
}
```

```java
// joi/10/juno/JFileNotFoundException.java (version 10)
//
//
// Copyright 1997-2001 Ethan Bolker and Bill Campbell

/**
 * The Exception thrown when a JFile isn’t found
 *
 * @version 10
 */

class JFileNotFoundException extends JunoException
{
    String jfilename;

    /**
     * Construct a new JFileNotFoundException
     *
     * @param jfilename the file sought
     */

    public JFileNotFoundException( String jfilename )
    {
        super( "JFile " + jfilename + " not found." );
        this.jfilename = jfilename;
    }

    /**
     * Get the name of the file that wasn’t there.
     *
     * @return the file name
     */

    public String getJFilename()
    {
        return jfilename;
    }
}
```

```java
// joi/10/juno/GUILoginConsole.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;

/**
 *  The graphical user interface to Juno.
 */

public class GUILoginConsole extends JFrame
    implements OutputInterface
{
    private static final int FIELDWIDTH = 30;
    private static final int FIELDHEIGHT = 5;

    private final Juno junoSystem;
    private WindowCloser closeMe; // to shut down Juno

    private String title;  // title for the windows

    // The interpreter interprets one-line commands.
    private InterpreterInterface interpreter;
    private boolean echoInput;

    // All output goes to messages.
    private JTextArea messages;

    /**
     *  Construct a GUI console for Juno.
     *
     *  @param title the title for this window.
     *  @param junoSystem the Juno system for which this is a GUI
     *  @param interpreter the object to which to send user input.
     *  @param echoInput true when input echoes to this console.
     */

    public GUILoginConsole( String title, Juno junoSystem,
                            InterpreterInterface interpreter,
                            boolean echoInput)
    {
        super( title );
        this.title       = title;
        this.junoSystem  = junoSystem;
        this.interpreter = interpreter;
        this.echoInput   = echoInput;
        this.closeMe = new WindowCloser( junoSystem );

        // Set up the look and feel;
        // Everthing is placed on a panel (using BorderLayout)

        JPanel panel = new JPanel();
```

```java
        panel.setLayout( new BorderLayout() );

        // First a tabbed pane, with two tabs:
        //   one for login, one for registration

        JTabbedPane tabs = new JTabbedPane();
        tabs.addTab( "Login" , null,
                     new LoginPane( interpreter, echoInput, closeMe ) );
        tabs.addTab( "Register", null,
                     new RegisterPane( interpreter, echoInput ) );
        tabs.setSelectedIndex( 0 );  // Login selected by default
        panel.add( tabs, BorderLayout.NORTH );

        // and the output messages area.
        panel.add( new JLabel( "Messages:" ), BorderLayout.CENTER );
        messages = new JTextArea( FIELDHEIGHT, FIELDWIDTH );
        panel.add( messages, BorderLayout.SOUTH );

        // Add the panel to this JFrame
        this.getContentPane().add( panel );

        // Closing this window
        this.setDefaultCloseOperation(JFrame.DO_NOTHING_ON_CLOSE);
        this.addWindowListener( closeMe );

        // Size and display this JFrame
        pack();
        show();
    }

    // Implementing the OutputInterface. Everything goes to the
    // single message area.

    /**
     * Write a String followed by a newline
     * to message area.
     *
     * @param str - the string to write
     */

    public void println(String str )
    {
        messages.append( str + "\n" );
    }

    /**
     * Write a String followed by a newline
     * to message area.
     *
     * @param str - the String to write
     */

    public void errPrintln(String str )
    {
        println( str );
    }

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if echoing input.
     */

    public boolean isEchoInput()
    {
        return echoInput;
    }

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if GUI
     */

    public boolean isGUI()
    {
        return true;
    }

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if remote
     */

    public boolean isRemote()
    {
        return false;
    }

    // The Login pane is specified in a private inner class,
    // visible only here.

    private class  LoginPane extends JPanel
    {
        // The login pane has two text fields and two buttons.
        private JTextField nameField;
        private JTextField passwordField;

        private JButton ok;
        private JButton exit;

        private WindowCloser closeMe; // to shut down Juno
        // Construct the login pane and set up its listeners.

        public LoginPane( InterpreterInterface interpreter,
                          boolean echoInput, WindowCloser closeMe )
        {
            super();
            this.closeMe = closeMe;

            // Set up the look and feel.
```

```java
            // Everything will go into a vertical Box, a container
            // whose contents are laid out using BoxLayout

            Box box = Box.createVerticalBox();

            // First a panel, containing the two text fields

            JPanel p  = new JPanel();
            p.setLayout( new GridLayout( 4 , 1 ) );

            p.add( new JLabel( "Login:" ) );
            nameField = new JTextField( FIELDWIDTH );
            p.add( nameField );

            p.add( new JLabel( "Password:" ) );
            passwordField = new JPasswordField( FIELDWIDTH );
            p.add( passwordField );
            box.add( p );
            box.add( Box.createVerticalStrut( 15 ) );

            // Then a horizontal Box containing the two buttons

            Box row = Box.createHorizontalBox();
            row.add( Box.createGlue() );

            ok = new JButton( "OK" );
            row.add( ok );
            row.add( Box.createGlue() );

            exit = new JButton( "Exit" );
            row.add( exit );
            row.add( Box.createGlue() );
            box.add( row );
            box.add( Box.createVerticalStrut( 15 ) );

            this.setLayout( new BorderLayout() );
            this.add( box, BorderLayout.CENTER );

            // Set up the listeners (the semantics)

            ok.addActionListener( new LoginProcessor() );
            exit.addActionListener( closeMe ); // shuts down Juno
        }

        // An inner inner class for the semantics
        // when the user clicks OK.

        private class LoginProcessor implements ActionListener
        {
            public void actionPerformed(ActionEvent e)
            {
                String str = nameField.getText() +  " " +
                    passwordField.getText();
                nameField.setText("");
                passwordField.setText("");
                messages.setText(str+'\n');  // for debugging
                interpreter.interpret( str );
            }
        }
    }

    // The Register pane is specified in a private inner class,
    // visible only here.

    private class  RegisterPane extends JPanel
    {
        // The register pane has four textfields and two buttons.
        private JTextField chosenName;
        private JTextField fullName;
        private JTextField password1;
        private JTextField password2;

        private JButton register;
        private JButton clear;

        public RegisterPane( InterpreterInterface interpreter,
                             boolean echoInput)
        {
            super();

            // Define the look and feel
            // Everything goes into a vertical Box
            Box box = Box.createVerticalBox();

            // First a panel containing the text fields

            JPanel p  = new JPanel();
            p.setLayout( new GridLayout( 0 , 1 ) );

            p.add( new JLabel( "Choose login name:" ) );
            chosenName = new JTextField( FIELDWIDTH );
            p.add( chosenName );

            p.add( new JLabel( "Give full name:" ) );
            fullName = new JTextField( FIELDWIDTH );
            p.add( fullName );

            p.add( new JLabel( "Choose password:" ) );
            password1  = new JPasswordField( FIELDWIDTH );
            p.add( password1 );

            p.add( new JLabel( "Retype password:" ) );
            password2  = new JPasswordField( FIELDWIDTH );
            p.add( password2 );

            box.add( p );
            box.add( Box.createVerticalStrut( 15 ) );

            // Then a horizontal Box containing the buttons

            Box row = Box.createHorizontalBox();
            row.add( Box.createGlue() );
```

```java
            register = new JButton( "Register" );
            row.add( register );
            row.add( Box.createGlue() );
            clear = new JButton( "Clear" );
            row.add( clear );
            row.add( Box.createGlue() );
            box.add( row );
            box.add( Box.createVerticalStrut( 15 ) );

            this.setLayout( new BorderLayout() );
            this.add( box, BorderLayout.CENTER );

            // Set up the listeners (the semantics)

            register.addActionListener( new Registration() );
            clear.addActionListener( new Cleanser() );
        }

        // An inner inner class for the semantics when the user
        // clicks Register.

        private class Registration implements ActionListener
        {
            public void actionPerformed(ActionEvent e)
            {
                if ( password1.getText().trim().equals(
                     password2.getText().trim() ) ) {
                    String str = "register " +
                        chosenName.getText() +  " " +
                        password1.getText() + " " +
                        fullName.getText() ;
                    chosenName.setText("");
                    fullName.setText("");
                    messages.setText(str+'\n');  // for debugging
                    interpreter.interpret(str);
                }
                else {
                    messages.setText(
                        "Sorry, passwords don't match.\n" );
                }
                password1.setText("");
                password2.setText("");
            }
        }

        // An inner inner class for the semantics when the user
        // clicks Clear.

        private class Cleanser implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                chosenName.setText("");
                fullName.setText("");
                password1.setText("");
                password2.setText("");
            }
        }
```

```
    }

    //  A WindowCloser instance handles close events generated
    //  by the underlying window system with its windowClosing
    //  method, and close events from buttons or other user
    //  components with its actionPerformed method.
    //
    //  The action is to shut down Juno.

    private static class WindowCloser extends WindowAdapter
        implements ActionListener
    {
        Juno system;

        public WindowCloser( Juno system )
        {
            this.system = system;
        }

        public void windowClosing (WindowEvent e)
        {
            this.actionPerformed( null );
        }

        public void actionPerformed(ActionEvent e)
        {
            if (system != null) {
                system.shutDown();
            }
            System.exit(0);
        }
    }

    /**
     * main() in GUILoginConsole class for
     * unit testing during development.
     */

    public static void main( String[] args )
    {
        new GUILoginConsole( "GUItest", null, null, true ).show();
    }
}
```

```java
// joi/10/juno/GUIShellConsole.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

/**
 *  The GUI to the Juno system Shell.
 */

public class GUIShellConsole extends JFrame
    implements OutputInterface
{
    private static final int FIELDWIDTH  = 50;
    private static final int FIELDHEIGHT = 10;

    // the components on the window

    private JLabel promptLabel = new JLabel();
    private JTextField commandLine = new JTextField( FIELDWIDTH );
    private JButton doIt       = new JButton( "Do It" );
    private JButton logout     = new JButton( "Logout" );
    private JTextArea stdout   =
                         new JTextArea( FIELDHEIGHT, FIELDWIDTH );
    private JTextArea stderr   =
                         new JTextArea( FIELDHEIGHT/2, FIELDWIDTH );

    private Shell sh;    // for interpreting shell commands
    private WindowCloser closeMe; // for logging out.

    private boolean echoInput;

    /**
     * Construct a GUI console for a shell.
     *
     * @param title the title to  display in the frame.
     * @param sh the shell to interpret commands.
     * @param echoInput is input to be echoed?
     */

    public GUIShellConsole( String title,
                            Shell sh,
                            boolean echoInput )
    {
        this.sh = sh;
        this.echoInput   = echoInput;

        setTitle( title );
        setPrompt( sh.getPrompt() );

        // set up console’s look and feel

```

```java
        JPanel outerPanel = new JPanel();
        outerPanel.setLayout( new BorderLayout() );

        Box box = Box.createVerticalBox();

        JPanel commandPanel = new JPanel();
        commandPanel.setLayout( new BorderLayout() );
        commandPanel.add( promptLabel, BorderLayout.NORTH );
        commandPanel.add( commandLine, BorderLayout.CENTER );
        box.add( commandPanel );
        box.add( Box.createVerticalStrut( 10 ) );

        Box buttons = Box.createHorizontalBox();
        buttons.add( Box.createGlue() );
        buttons.add( doIt );
        buttons.add( Box.createGlue() );
        buttons.add( logout );
        buttons.add( Box.createGlue() );
        box.add( buttons );
        box.add( Box.createVerticalStrut( 10 ) );

        JPanel stdoutPanel = new JPanel();
        stdoutPanel.setLayout( new BorderLayout() );
        stdoutPanel.add( new JLabel( "Standard output:"),
                         BorderLayout.NORTH );

        stdoutPanel.add( new JScrollPane( stdout ),
                         BorderLayout.CENTER );

        box.add( stdoutPanel );
        box.add( Box.createVerticalStrut( 10 ) );
        stdout.setEditable( false );

        JPanel stderrPanel = new JPanel();
        stderrPanel.setLayout( new BorderLayout() );
        stderrPanel.add( new JLabel( "Error output:" ),
                         BorderLayout.NORTH );
        stderrPanel.add( new JScrollPane( stderr ),
                         BorderLayout.CENTER );
        box.add( stderrPanel );
        box.add( Box.createVerticalStrut( 10 ) );
        stderr.setEditable( false );

        outerPanel.add( box, BorderLayout.CENTER );
        this.getContentPane().add( outerPanel, BorderLayout.CENTER );

        // Install menus and tool bar.

        JMenuBar menuBar  = new JMenuBar();
        JMenu commandMenu = new JMenu( "Command" );
        JMenu helpMenu    = new JMenu( "Help" );

        JToolBar toolBar  = new JToolBar();

        // Create menu items and tool buttons for each shell command

        ShellCommandTable table = sh.getSystem().getCommandTable();
        String [] commandNames = table.getCommandNames();
        for ( int i = 0; i < commandNames.length; i++ ) {

            String commandName = commandNames[i];
            ShellCommand command =
                table.lookup( commandName );

            CommandMenuAction commandAction =
                new CommandMenuAction( commandName,
                                       command.getArgString() );

            HelpMenuAction helpAction =
                new HelpMenuAction( commandName,
                                    command.getArgString(),
                                    command.getHelpString() );

            JMenuItem item1 = commandMenu.add( commandAction );
            JMenuItem item2 = helpMenu.add( helpAction );
            JButton button  = toolBar.add( commandAction );
            button.setToolTipText( command.getHelpString() );
        }

        this.setJMenuBar( menuBar );
        this.getContentPane().add( toolBar,
                                   BorderLayout.NORTH );
        menuBar.add( commandMenu );
        menuBar.add( helpMenu );

        pack();
        show();

        // add Listener to the Do It button

        doIt.addActionListener( new Interpreter() );

        // add listener to the Logout button and window closer

        closeMe = new WindowCloser( this );
        logout.addActionListener( closeMe );
        this.addWindowListener( closeMe );
    }

    // Set the GUI prompt

    private void setPrompt(String prompt)
    {
        this.promptLabel.setText(prompt);
    }

    // Implementing the OutputInterface.
    // Everything goes to the single message area.

    public void println( String str )
    {
        stdout.append(str + "\n");
```

```java
    }

    public void errPrintln( String str )
    {
        stderr.append(str + "\n");
    }

    public boolean isGUI()
    {
        return true;
    }

    public boolean isRemote()
    {
        return false;
    }

    public boolean isEchoInput()
    {
        return echoInput;
    }

    // An inner class for the semantics when the user submits
    // a ShellCommand for execution.

    private class Interpreter
        implements ActionListener
    {
        public void actionPerformed( ActionEvent e )
        {
            String str = commandLine.getText();
            stdout.append( sh.getPrompt() + str + '\n');
            if (sh.interpret( str )) {
                setPrompt( sh.getPrompt() );
            }
            else {
                closeMe.actionPerformed(null);
            }
        }
    }

    private class CommandMenuAction extends AbstractAction
    {
        private String argString;
        private String helpString;

        public CommandMenuAction( String text, String argString )
        {
            super( text );
            this. argString = argString;
        }

        public void actionPerformed( ActionEvent e )
        {
            commandLine.setText( getValue( Action.NAME ) +
                                 " " + argString );
        }
    }

    private class HelpMenuAction extends AbstractAction
    {
        private String argString;
        private String helpString;

        public HelpMenuAction( String text, String argString,
                               String helpString )
        {
            super( text );
            this. argString = argString;
            this.helpString = helpString;
        }

        public void actionPerformed( ActionEvent e )
        {
            stdout.append( getValue( Action.NAME ) + ": " +
                           helpString );
        }
    }

    //  A WindowCloser instance handles close events generated
    //  by the underlying window system with its windowClosing
    //  method, and close events from buttons or other user
    //  components with its actionPerformed method.
    //
    //  The action is to logout and dispose of this window.

    private static class WindowCloser extends WindowAdapter
        implements ActionListener
    {
        Frame myFrame;

        public WindowCloser( Frame frame ) {
            myFrame = frame;
        }

        public void windowClosing (WindowEvent e)
        {
            this.actionPerformed( null );
        }

        public void actionPerformed(ActionEvent e)
        {
            myFrame.dispose();
        }
    }
}
```

```java
// joi/10/juno/InterpreterInterface.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * Juno needs an interpreter to process the user’s response to
 * the login: prompt (or what she enters on a GUILoginConsole).
 *
 * Each Shell needs an interpreter for shell command lines,
 * whether entered with a GUI or a CLI.
 *
 * @version 10
 */

public interface InterpreterInterface
{
    /**
     * Interpret a command line String.
     *
     * @param str the String to interpret
     * @return true, unless str tells you there’s nothing to follow
     */

    public boolean interpret( String str );
}
```

```java
// joi/10/juno/InputInterface.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 *  Juno consoles use the same abstract method
 *  for input, so it is specified here.
 */

public interface InputInterface
{
   /**
    * Read a line (terminated by a newline).
    *
    * @param promptString output string to prompt for input
    * @return the string (without the newline character)
    */

   public String readLine( String promptString );
}
```

```java
// joi/10/juno/OutputInterface.java
//
//
// Copyright 2003 Ethan Bolker and Bill Campbell

/**
 *  All Juno consoles use the same abstract methods
 *  for output, so they are specified here.
 */

public interface OutputInterface
{
    /**
     * Write a String followed by a newline
     * to console output location.
     *
     * @param str - the string to write
     */

    public void println(String str );

    /**
     * Write a String followed by a newline
     * to console error output location.
     *
     * @param str - the String to write
     */

    public void errPrintln( String str );

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if echoing input.
     */

    public boolean isEchoInput();

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if GUI
     */

    public boolean isGUI();

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if remote
     */

    public boolean isRemote();
}

```

```java
// joi/10/juno/JunoTerminal.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

/**
 * A Command line interface terminal for Juno.
 *
 * @version 10
 */

public class JunoTerminal
    implements InputInterface, OutputInterface
{
    private Terminal terminal; // the delegate terminal
    private boolean  echo;     // are we echoing input?

    /**
     * Construct a JunoTerminal
     *
     * Allows for input echo, when, for example, input is redirected
     * from a file.
     *
     * @param echo whether or not input should be echoed.
     */

    public JunoTerminal( boolean echo )
    {
        this.echo = echo;
        terminal = new Terminal( echo );
    }

    // Implement InputInterface

   /**
    * Read a line (terminated by a newline).
    *
    * @param promptString output string to prompt for input
    * @return the string (without the newline character)
    */

    public String readLine( String promptString )
    {
        return terminal.readLine( promptString );
    }

    // Implement OutputInterface

   /**
     * Write a String followed by a newline
     * to console output location.
     *
     * @param str - the string to write
     */

    public void println(String str )
```

```java
    {
        terminal.println( str );
    }

    /**
     * Write a String followed by a newline
     * to console error output location.
     *
     * @param str - the String to write
     */

    public void errPrintln(String str )
    {
        terminal.errPrintln( str );
    }

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if echoing input.
     */

    public boolean isEchoInput()
    {
        return echo;
    }

    /**
     * Query what kind of console this is.
     *
     * @return false, since it is not a GUI
     */

    public boolean isGUI()
    {
        return false;
    }

    /**
     * Query what kind of console this is.
     *
     * @return false, since it is not remote.
     */

    public boolean isRemote()
    {
        return false;
    }
}
```

```java
// joi/10/juno/RemoteConsole.java
//
//
// Copyright 2003 Bill Campbell and Ethan Bolker

import java.io.*;
import java.net.*;
import java.util.*;
import java.text.*;

/**
 * A remote console listens on a port for a remote login to
 * a running Juno system server.
 *
 * @version 10
 */

public class RemoteConsole extends Thread
    implements OutputInterface, InputInterface
{
    // default just logs connection start and end
    // change to true to log all i/o
    private static boolean logall = false;

    private Juno system;
    private boolean echo;
    private InterpreterInterface interpreter;

    private Socket clientSocket;
    private BufferedReader in;
    private PrintWriter out;
    private int sessionCount = 0;

    private PrintWriter junoLog;

    /**
     * Construct a remote console to listen for users trying
     * to connect to Juno.
     *
     * Called from Juno main.
     *
     * @param system the Juno system setting up this console.
     * @param echo whether or not input should be echoed.
     * @param port the port on which to listen for requests.
     */

    public RemoteConsole( Juno system, boolean echo, int port )
    {
        this.echo = echo;
        Date now  = new Date();
        junoLog   = openlog(now);
        log("*** Juno server started " + now + "\n");
        try {
            ServerSocket ss = new ServerSocket(port);
            while (true) {
                clientSocket = ss.accept();
```

```java
                new RemoteConsole( system, echo, clientSocket,
                                   junoLog, ++sessionCount).start();
            }
        }
        catch (IOException e) {
            System.out.println("Remote login not supported");
            System.exit(0);
        }
        finally {
            system.shutDown();
        }
    }

    /**
     * Construct a remote console for a single remote user.
     *
     * @param system the Juno system to which the user is connecting.
     * @param echo whether or not input should be echoed.
     * @param clientSocket the socket for the user’s connection
     * @param junoLog track all user i/o
     * @param sessionCount this session’s number
     */

    public RemoteConsole( Juno system, boolean echo, Socket clientSocket,
                          PrintWriter junoLog, int sessionCount )
    {
        this.system = system;
        this.echo = echo;
        this.clientSocket = clientSocket;
        this.junoLog = junoLog;
        this.sessionCount = sessionCount;
    }

    /**
     * Action when the thread for this session starts.
     */

    public void run()
    {
        log("*** " + sessionCount + ", " +
            clientSocket.getInetAddress()  + ", "  +
            new Date());
        try {
            setUpConnection();
            String s = this.readLine
                ("Please sign the guest book (name, email): ");
            this.println("Thanks, " + s);
            if (!logall) {
                log("guest book: " + s);
            }
            new LoginInterpreter( system, this ).CLILogin();
            clientSocket.close();
        }
        catch (IOException e) {
            log("*** Error " + e);
        }
        log("*** end session " + sessionCount);
    }

    /**
     * Create the readers and writers for the socket
     * for this session.
     */

    private void setUpConnection()
         throws IOException
    {
        in = new BufferedReader(
                new InputStreamReader(clientSocket.getInputStream()));
        out = new PrintWriter(
                new OutputStreamWriter(clientSocket.getOutputStream()));
    }

    // implement the InputInterface

   /**
    * Read a line (terminated by a newline) from console socket.
    *
    * Log the input line before returning it if required.
    *
    * @param promptString output string to prompt for input
    * @return the string (without the newline character)
    */

    public String readLine( String promptString )
    {
        String s = "";
        this.print(promptString);
        out.flush();
        try {
            s = in.readLine();
            if (logall) {
                log("> " + s);
            }
            if (echo) {
                out.println(s);
            }
        }
        catch (IOException e) {
            String msg = "IO error reading from remote console";
            System.out.println(msg);
            out.println(msg);
        }
        return s;
    }

    /**
     * Write a String to console socket.
     *
     * Log the output if required.
     *
     * @param str - the string to write
```

```java
     */

    public void print( String str )
    {
        out.print( str );
        out.flush();
        if (logall) {
            log("< " + str + "\\");
        }
    }

    // implement the OutputInterface

    /**
     * Write a String followed by a newline
     * to console socket.
     *
     * Log the output if required.
     *
     * @param str - the string to write
     */

    public void println( String str )
    {
        out.println( str + '\r' );
        out.flush();
        if (logall) {
            log("< " + str);
        }
    }

    /**
     * Write a String followed by a newline
     * to console error output location. That's
     * just the socket.
     *
     * @param str - the String to write
     */

    public void errPrintln(String str )
    {
        println( str );
    }

    /**
     * Query what kind of console this is.
     *
     * @return false since it's not a GUI.
     */

    public boolean isGUI()
    {
        return false;
    }

    /**
```

```
     * Query what kind of console this is.
     *
     * @return true since it is remote.
     */

    public boolean isRemote()
    {
        return true;
    }

    /**
     * Query what kind of console this is.
     *
     * @return true if and only if echoing input.
     */

    public boolean isEchoInput()
    {
        return echo;
    }

    /**
     * Log a String.
     *
     * @param str the String to log.
     */

    private void log(String str)
    {
        junoLog.println(sessionCount + ": " + str);
        junoLog.flush();
    }

    /**
     * Open a log for this console.
     *
     * @param now the current Date.
     */

    private PrintWriter openlog(Date now)
    {
        PrintWriter out = null;
        SimpleDateFormat formatter
            = new SimpleDateFormat ("MMM.dd:hh:mm:ss");
        String dateString = formatter.format(now);
        String filename = "log-" + dateString;
        try { out = new PrintWriter(
                    new BufferedWriter(
                    new FileWriter(filename)));
        }
        catch (Exception e) {
            out = new PrintWriter(new FileWriter(FileDescriptor.out));
        }
        return out;
    }
}
```