/**
 * 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     * @param amount the amount to be withdrawn.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.
     */
    public int getBalance() {
        return balance;
    }
}

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"

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

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 );
}

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();
}
}

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 { // account
int action = atm.readInt("Valid actions: deposit, withdraw, balance, exit. 
Select an action: ");
Account transactionTarget = account.transactions[ action ];
switch ( transactionTarget ) {
    case deposit:
        account.deposit( atm.readFloat("Deposit: ") );
        break;
    case withdraw:
        account.withdraw( atm.readFloat("Withdraw: ") );
        break;
    case balance:
        atm.println("Balance: " + account.balance() );
        break;
    case exit:
        atm.println("Exiting.");
        moreTransactions = false;
        break;
    default:
        atm.println("Invalid action.");
        break;
}
}
}
```
}
```java
class Bank {
    // ... (same as provided) ...
}

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

This code snippet demonstrates a simple Java Bank simulation. The `main` method initializes a `Bank` object and opens it for transactions. The `Bank` class encapsulates the logic for handling `deposit`, `withdraw`, and `balance` commands from a user.

The provided code includes the necessary method signatures and logic to handle these transactions. The `atm.readInt` method is used to read an integer from the user, and the `account.deposit`, `account.withdraw`, and `account.getBalance` methods are called to perform the respective operations on the bank account.

The `javaBank.open()` method likely initializes the bank environment, possibly setting up the user interface or initializing the state of the bank account.

Overall, this code is a basic example of how to simulate a bank using object-oriented programming in Java, focusing on user interaction and account management.
// 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()
{
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();
}
113// A ShutDownLight instance handles close events generated
114// by the underlying window system with its windowClosing
115// method.
116//
117// This is an inner class, declared inside the
118// TrafficLight class since it's used only here.
119
120private static class ShutDownLight extends WindowAdapter
121{
122// Close the window by shutting down the light.
123
124public void windowClosing (WindowEvent e)
125{
126    System.exit(0);
127}
128}
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
* the action performed when a push of the button is detected:
*/

public void actionPerformed( ActionEvent event )
{
this.sequencer.next();
}
// 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:
                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?!" );
        }
    }
}
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.
 */

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

/**
 * Darken this Lens.
 */

public void turnOff()
{
currentColor = offColor;
this.repaint();
}
}
/**
 * An EStore object simulates the behavior of a simple online 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 Mall";

// 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();
int totalCost = basket.getCost();
browser.println("We are shipping " + numberPurchased + " Items 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
List 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;
    }
}

Apr 15 21:52 2004  Listing 1.8 Item.java Page 1
/**
 * 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 += 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.
     *
     * @return the total cost.
     */
    public int getCost()
    {
        return cost;
    }
}