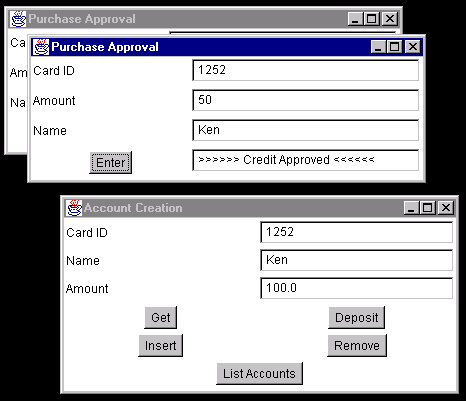
**Applications with Multiple Windows**

***Introduction***

Creating an application with multiple views or windows is straightforward. For example, consider a simplified credit card approval system. The application allows users to:

|  | create new accounts |
| --- | --- |
|  | delete old accounts |
|  | list all accounts |
|  | make deposits to accounts |
|  | request approval for credit card purchases against accounts. |

When the application approves a purchase, the amount is deducted from the account indicated by the credit card. The application's interface consists of one window for creating and managing accounts and any number of other windows for approving purchases. Figure 1 shows the application running in three windows. The manager has just entered a new account for Ken with a balance of $100.00. Ken then requests approval for a $50.00 purchase, and the request is approved.



**Figure 1** The user interface for the credit approval program

***Digression into Reality***

The application as described above is not a realistic credit card approval system. Rather it is intended to illustrate the use of multiple windows running on a single computer. After all it does not really make sense for the account creation window and the purchase approval windows to be running on the same computer; however, a more realistic system can be built fairly easily by anyone with a knowledge of Java sockets and threads. In this more realistic system:

* ∙         The account creation window runs 24 hours a day on a computer at a bank's headquarters and listens for connection requests on an agreed upon socket.
* ∙         Meanwhile retail merchants use Internet browsers to run applets corresponding to purchase approval windows. When a merchant clicks the **Enter** button in a purchase approval applet, the applet makes a connection to the account creation program via the agreed upon socket.
* ∙         The account creation program then spawns a thread that swaps information with the applet over a new socket created on the fly, and the account creation program goes back to listening for connection requests from other applets. The connection between the spawned thread and the applet is dropped when the exchange of information has been completed.

***The Classes***

But now let us return to the unreal world of our application. The application consists of the following classes:

| **Class** | **Responsibility** |
| --- | --- |
| CreditAccountsApplication | This class consists solely of the static method main. The method main instantiates the model and then instantiates the various views, passing to each a reference to the model. Finally, main opens the views so that the user can start using the application. |
| CreditAccountsModel | This class manages the collection of user accounts. |
| AccountCreationView | This class handles the window for creating and managing accounts. |
| CreditApprovalView | This class handles the window for approving purchases. |
| Account | This class represents an individual account. |

***The Implementation***

We start with the code for the CreditAccountsApplication:

// CreditAccountsApplication

// (c) 1999 Ken Lambert and Martin Osborne

import java.awt.\*;

public class CreditAccountsApplication extends Object{

public static void main (String[] args){

CreditAccountsModel model = new CreditAccountsModel();

Frame frm = new CreditApprovalView(model);

frm.setSize (400, 150);

frm.setLocation (0, 0);

frm.setVisible(true);

frm = new CreditApprovalView(model);

frm.setSize (400, 150);

frm.setLocation (100, 155);

frm.setVisible(true);

frm = new AccountCreationView(model);

frm.setSize (400, 200);

frm.setLocation (200, 310);

frm.setVisible(true);

}

}

The model class, CreditAccountsModel, is simple. We use a Map to store the collection of accounts:

// CreditAccountsModel

// (c) 1999 Ken Lambert and Martin Osborne

import java.util.\*;

public class CreditAccountsModel extends Object{

private Map map;

public CreditAccountsModel(){

map = new HashMap();

}

public Account getAccount (String accountId){

return (Account)(map.get (accountId));

}

public boolean insertAccount (Account account){

if (map.containsKey (account.getCardId()))

return false;

else{

map.put (account.getCardId(), account);

return true;

}

}

public void removeAccount (Account account){

map.remove (account.getCardId());

}

public String toString(){

Iterator iter = map.values().iterator();

String str = "";

while (iter.hasNext())

str += iter.next() + "\n";

return str;

}

}

The view classes access accounts via the model. The CreditApprovalView is the simpler of the two views. When the **Enter** button is clicked, the view communicates with the model in order to determine if there is enough money in the account to cover the purchase. Here is the code for the class CreditApprovalView:

// CreditApprovalView

// (c) 1999 Ken Lambert and Martin Osborne

import java.awt.\*;

import BreezyGUI.\*;

public class CreditApprovalView extends GBFrame{

// Window objects --------------------------------------

Label lbCardId = addLabel ("Card ID",1,1,1,1);

Label lbPurchaseAmount = addLabel ("Amount",2,1,1,1);

Label lbName = addLabel ("Name",3,1,1,1);

TextField flCardId = addTextField ("",1,2,1,1);

DoubleField flPurchaseAmount = addDoubleField (0,2,2,1,1);

TextField flName = addTextField ("",3,2,1,1);

Button lastButton = addButton ("Enter",4,1,1,1);

TextField flApproval = addTextField ("",4,2,1,1);

// Other instance variables ---------------------------------

private CreditAccountsModel model;

// Constructor-----------------------------------------------

public CreditApprovalView(CreditAccountsModel model){

setTitle ("Purchase Approval");

flName.setEditable (false);

flApproval.setEditable (false);

this.model = model;

}

// buttonClicked method--------------------------------------

public void buttonClicked (Button buttonObj){

String cardId = flCardId.getText().trim();

double amount = flPurchaseAmount.getNumber();

Account account;

if (cardId.equals ("") || amount <= 0){

messageBox ("Enter the card id and a positive amount");

return;

}

account = model.getAccount (cardId);

if (account == null){

messageBox ("Invalid card id");

return;

}

flName.setText (account.getName());

if (account.withdraw (amount))

flApproval.setText (">>>>>> Credit Approved <<<<<<");

else

flApproval.setText (">>>>>> CREDIT DENIED <<<<<<");

}

}

The AccountCreationView is listed next:

// AccountCreationView

// (c) 1999 Ken Lambert and Martin Osborne

import java.awt.\*;

import BreezyGUI.\*;

public class AccountCreationView extends GBFrame{

// Window objects --------------------------------------

Label lbCardId = addLabel ("Card ID" ,1,1,1,1);

Label lbName = addLabel ("Name" ,2,1,1,1);

Label lbAmount = addLabel ("Amount" ,3,1,1,1);

TextField flCardId = addTextField ("" ,1,2,1,1);

TextField flName = addTextField ("" ,2,2,1,1);

DoubleField flAmount = addDoubleField (0 ,3,2,1,1);

Button btGet = addButton ("Get" ,4,1,1,1);

Button btInsert = addButton ("Insert" ,5,1,1,1);

Button btDeposit = addButton ("Deposit" ,4,2,1,1);

Button btRemove = addButton ("Remove" ,5,2,1,1);

Button btList = addButton ("List Accounts" ,6,1,2,1);

// Other instance variables ---------------------------------

private CreditAccountsModel model;

// Constructor-----------------------------------------------

public AccountCreationView(CreditAccountsModel model){

setTitle ("Account Creation");

this.model = model;

}

// buttonClicked method--------------------------------------

public void buttonClicked (Button buttonObj){

if (buttonObj == btGet)

get();

else if (buttonObj == btInsert)

insert();

else if (buttonObj == btDeposit)

deposit();

else if (buttonObj == btRemove)

remove();

else

listAccounts();

}

// private methods--------------------------------------

private void get(){

Account account = getAccount();

if (account == null)

return;

else

display (account);

}

private Account getAccount(){

String id = flCardId.getText().trim();

if (id.equals ("")){

messageBox ("Enter the card id first");

return null;

}

Account account = model.getAccount (id);

if (account == null){

messageBox ("There is no such account");

return null;

}

return account;

}

private void display (Account account){

if (account == null){

flCardId.setText ("");

flName.setText ("");

flAmount.setNumber (0);

}else{

flCardId.setText (account.getCardId());

flName.setText (account.getName());

flAmount.setNumber (account.getBalance());

}

}

private void insert(){

Account account = buildAccountFromScreenData();

if (account == null)

return;

if (!model.insertAccount (account)){

messageBox ("The account is already present");

return;

}

}

private Account buildAccountFromScreenData(){

String id = flCardId.getText().trim();

String name = flName.getText().trim();

double amount = flAmount.getNumber();

if (id.equals ("") || name.equals ("")){

messageBox ("Card id and name are both required");

return null;

}

if (amount < 0){

messageBox ("The amount must be nonnegative");

return null;

}

return new Account (id, name, amount);

}

private void deposit(){

Account account = getAccount();

if (account == null)

return;

double amount = flAmount.getNumber();

if (amount <= 0){

messageBox ("The deposit amount must be positive");

return;

}

account.deposit (amount);

flAmount.setNumber (account.getBalance());

}

private void remove(){

Account account = getAccount();

if (account == null)

return;

model.removeAccount (account);

display (null);

}

private void listAccounts(){

messageBox ("Here is a list of all the accounts\n" + model);

}

}

Finally, here is the code for the Account class:

// Account

// (c) 1999 Ken Lambert and Martin Osborne

public class Account extends Object{

private String cardId;

private String name;

private double balance;

public Account()

{

cardId = "";

name = "";

balance = 0;

}

public Account (String cardId, String name, double balance)

{

this.cardId = cardId;

this.name = name;

this.balance = balance;

}

public String getCardId ()

{

return cardId;

}

public String getName ()

{

return name;

}

public double getBalance ()

{

return balance;

}

public void deposit (double amount)

{

balance += amount;

}

public boolean withdraw (double amount)

{

if (amount > balance)

return false;

else{

balance -= amount;

return true;

}

}

public String toString()

{

return cardId + " " + name + " " + balance;

}

}