Bank Loan Sample Program

This sample bank loan program consists of several files that belong to the banking package. For this reason, all class files must be located within a directory named banking. Follow these steps to use the program.

1. Create a directory called banking.
2. Copy the `java` and `class` files into the banking directory.

For convenience, you might want to setup an environment variable for your sample class path. For example, on AIX, where the fully qualified path name for your banking directory might be `$JAVAHOME/devcon/samples/classes/banking`, you might set your sample class path name as follows:

```
export MYJSAMPLES=$JAVAHOME/devcon/samples/classes
```

where `JAVAHOME` is set to the directory where the Java Developer's Kit (JDK) is located.

Note: Since the banking loan application is in a package called banking, the Java runtimes expect your class to be prefaced with the package name, for example, banking. See step 4 below.

3. Follow these steps to run the sample program.

   - Unpack the files as appropriate.
   - Set up the appropriate environment variable for your JDK executables, such as `java` and `javac`. On AIX, for example,

     ```
     export PATH=$PATH:$JAVAHOME/bin
     ```

     where `JAVAHOME` is set to the directory in which the JDK is located.
   - Set up the appropriate environment variable for your JDK classes and for this sample (see step 2). Setting `CLASSPATH` for the first time on AIX by following step 2 in setting up an environment variable for your sample classes,

     ```
     export CLASSPATH=$JAVAHOME/classes:/$MYSAMPLES
     ```

     where `JAVAHOME` is set to the directory in which the JDK is located and `$MYSAMPLES` is set to the directory that contains the banking directory.

Note: Do not include banking in the path because that information is presented by the package prefaces included in your code or on the command line.

4. Run the java command as follows: `java banking.Job`.

Note: You must preface the Job class with the package name. The files in this package include:

- `Account.java`
- `BankJob.java`
- `InputManager.java`
- `Job.java`
- `MyMath.java`
- `ProcessAccount.java`

- Transaction.java
- Worker.java
- WorkItem.java

The classes in this package include:

- Account.class
- Approver.class
- BankJob.class
- FinishWork.class
- InputManager.class
- Job.class
- MyMath.class
- ProcessAccount.class
- ReadAccount.class
- Transaction.class
- Worker.class
- WorkItem.class

How the Classes Work Together

The following sections describe how the classes work together.

Job Class

- Instantiated by the `java application` command
- Processes standard input-optional burdens for the worker threads
- Creates the `BankJob` object and calls the `BankJob` run method

BankJob Object

- Creates an `InputManager` object and registers as an Observer with the `InputManager` object created
- Waits for the notification from the `InputManager` that the input is complete
- Creates an `Approver` object and invokes its approve method

Approver Object

- Creates the `Worker` thread objects
- Creates a `ReadAccount` object for each account the person owns
- Creates a `FinishWork` object with a release count equal to the number of `WorkItems` already created. When the other `WorkItems` are done, the `FinishWork` item cleans up
- Waits for the threads to finish
- Executes the loan approval algorithm

Worker Thread Object

- Gets a `WorkItem` off its work queue or steals a `WorkItem` from another worker's queue
- Invokes the `WorkItem`'s run method
- Decrease the release count for the `FinishWork` objects

WorkItems

- Execute the various work-read account transactions and update current balances

Note: If you compile this code, you will get a warning message that the `PrintStream` class has been depreciated.

BankJob.java

The BankJob class holds the main logic for processing the bank loan. The BankJob object is created by the Job object—this application's main method.

Other sample Java classes in this file include:

- Approver
- ReadAccount
- FinishWork (subclasses of WorkItem)

Sample Java classes used from other files include:

- InputManager
- Account
- Worker
- WorkItem
- MyMath
- ProcessAccount

Java classes used include:

- Date
- File
- Math
- Thread

Java interfaces used include:

- Observer

Features of the sample code include:

- Implementation of the Observer interface
- Threads—creation, join
- Error handling—try, catch (InterruptedException, StringIndexOutOfBoundsException)
- Alternative to global variables—static/final

This application uses the work-stealing algorithm described by Blumofe and Leiserson in "Scheduling Multithreaded Computations by Work Stealing." In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science* (FOCS). Santa Fe, NM. Nov 20-22, 1994. Pages 356-368.

Written by Peg MacPhail, IBM Technical Consultant

to you. IBM shall not be liable for any damages you suffer as a result of using, modifying, or distributing the sample program or its derivatives.

Each copy of any portion of this sample program or any derivative work, must include the above copyright notice and disclaimer of warranty.

BankJob.java

```
package banking;

import banking.InputManager;
import banking.MyMath;
import banking.ProcessAccount;
import banking.Worker;
import banking.WorkItem;
import java.util.*;
import java.io.*;

// This class implements the Observer interface and registers for
// notification by the InputManager which the BankJob object creates.
public class BankJob implements Observer {

  // The following are like global variables for the application
  // Note the use of modifiers "static" and "final"
  protected static final int WORKNUMBER=50;  // Max work chunk for workers
  protected static final int LARGEST=10000;  // Largest transaction amount
  protected static final int COUNT=5;        // Number of accounts
  protected static final int NUMTHREADS = 5; // Number of threads
  protected static InputManager IM;
  protected static Account [] accounts = new Account [COUNT];
  protected static Worker [] workers = new Worker [NUMTHREADS];
  protected static Thread [] threads = new Thread [NUMTHREADS];
  protected static FinishWork ender;         // Ending work item
  protected static String [] acctNames = new String [COUNT];
  protected static String [] acctTypes =
        {"checking","savings", "loan", "IRA", "transactions"};
  //protected static int [] no = {200, 100, 50, 50, 100};
  protected static int [] no = {2, 1, 1, 1, 1};  // Transactions
  protected static int readBurden;           // Artificial load on workers
  protected static int processBurden;        // Artificial load on workers
  protected static boolean workDone;
  protected static boolean inputDone;

  // These variables are for the BankJob object, only
  private static Thread input;
  private static String name;
  private static double amount;

  // Constructor receiving artificial burdens for the worker threads -
  // rb and pb
  public BankJob(int rb, int pb) {
    readBurden = rb;
    processBurden = pb;
    IM = new InputManager();     // InputManager extends Observable class
    IM.addObserver(this);        // BankJob implements Observer
```

```
    input = new Thread(IM);
    workDone = false;
    inputDone = false;
  }

  // Main routine for the BankJob - called by Job object after the BankJob
  // object is created
  public void runJob()  {
    input.start();
    Date begin, end;
    Approver apper;

    while (!workDone) {
      while (!inputDone) {
       waitOnInput();
      }

      IM.appendText("Finished building database.\n");

      begin = new Date();

      apper = new Approver(name, amount, this);

      if (apper.approveIt()) {
      IM.appendText("Loan of " + IM.formatDollars(amount) +
        " approved for " + name + ".  Congratulations!!\n");
      }
      else {
      IM.appendText("Loan of " + IM.formatDollars(amount) +
        " NOT approved for " + name  + ".  Too bad!!\n");
      }

      end = new Date();
      IM.appendText("Done.\n" + printElapsedTime(begin,new Date()));

      inputDone = false;
      IM.reset();
    }
    try {
      input.join();
    }
    catch (InterruptedException e) {
      IM.appendText("Thread interrupt" + e);
    }
  }

  public static String printElapsedTime(Date begin, Date end) {
    long timer = end.getTime() - begin.getTime();
    long front = (long) Math.floor(timer/1000);
    long back = timer - front;
    return ("Time elapsed = " + front + "." + back + " seconds\n");
  }

  public synchronized void waitOnInput() {
    try {
      wait(5000);
    }
    catch (InterruptedException e) {}
  }
```

```
  // This method is invoked by the InputManager when it notifys Observers
  public void update(Observable o, Object arg) {
    if (arg.equals("Ready")) {
      name = IM.getName();
      String fileName = name.replace(' ','M');
      if (fileName.length() > 7) {
      try {
        fileName = fileName.substring(0,7);
      }
      catch (StringIndexOutOfBoundsException e) {}
      }
      amount = IM.getAmount();
      ender = null;
      // Account names are based on the loan applicant's name
      acctNames [0] = fileName+"c";
      acctNames [1] = fileName+"s";
      acctNames [2] = fileName+"l";
      acctNames [3] = fileName+"I";
      acctNames [4] = "transactions";

      buildIt(COUNT, LARGEST);
      inputDone = true;
    }
  }

  // This method with setAccount creates dummy account files
  public void buildIt(int num, int largest) {
    Account a [] = new Account [num];
    for (int i=0;i<num;i++) {
      a [i] = setAccount (acctNames[i], acctTypes[i], no[i], largest);
      a[i].writeOut();
    }
  }

  public Account setAccount (String name, String type, int num, int largest)
{
    Account  a = new Account(name, type, this);
    for (int i=0;i<num;i++) {
      double amt = (double) MyMath.getRandomNumber(largest*100)/100;
      if ( MyMath.getRandomNumber(4) == 2) amt = -amt;
      a.postTransaction(new Transaction (name,amt));
    }
    return a;
  }
}

// This class is a work item for reading the account data
class ReadAccount extends WorkItem {
  ReadAccount(Worker wm, int i, int rc, BankJob j) {
    super(wm, i, rc, "read", j);
    checkToPost();
  }
  public void run () {
   // Read in information for this account
   job.accounts [index] =
          new Account(job.acctNames[index],job.acctTypes[index],job);
   job.accounts [index].readIn();
   // Create a ProcessAccount work item for this account
   ProcessAccount a =
```

```
            new ProcessAccount(workMaster, index, 0, job, 0,
              (job.accounts [index].getTransCount()-1));
  }
}

// This class is a work item for ending the work on the accounts
class FinishWork extends WorkItem {
  FinishWork (Worker wm, int rc, BankJob j) {
    super(wm, 0, rc, "finish", j);
    checkToPost();
  }
  public void run () {
    for (int i=0;i<job.NUMTHREADS;i++) {
      if (job.workers[i] != workMaster) {
      job.workers[i].stop();
      }
    }
    workMaster.stop();
  }
}

// This class implements the approval algorithm for this application
class Approver {
  private String name;
  private double amount;
  private static double loan_factor = 0.15;
  private BankJob job;
  Approver(String n, double amt, BankJob j) {
   name = n;
   amount = amt;
   job = j;
   job.IM.appendText("Getting approval for loan for " + name +
              " for " + job.IM.formatDollars(amount) + "\n");
  }

  public boolean approveIt() {
    for (int i=0;i<job.NUMTHREADS;i++) {
      job.workers [i] = new Worker (i, job);
    }
    job.ender = new FinishWork(job.workers [0], 1, job);
    for (int i=0;i<job.COUNT;i++) {
      new ReadAccount(job.workers [0], i, 0, job);
    }
    job.ender.decreaseRC();
    for (int i=0;i<job.NUMTHREADS;i++) {
      job.threads [i] = new Thread (job.workers [i]);
      job.threads [i].setName("worker" + String.valueOf(i));
      job.threads [i].start();
    }
    for (int i=0;i<job.NUMTHREADS;i++) {
      try {
      job.threads [i].join();
      }
      catch (InterruptedException e) {
      job.IM.appendText("Thread interupt" + e);
      }
    }
    job.IM.appendText("Current balances for " + name + " are:\n");
    for (int i=0;i<job.COUNT-1;i++) {
```

```
      String a = job.IM.formatDollars(job.accounts[i].getAccountBalance());
      job.IM.appendText("   " + job.accounts[i].getAccountType() +
            " balance is " + a + "\n");
    }
    boolean approve = false;
    if (amount < (loan_factor *(job.accounts[0].getAccountBalance() +
                        job.accounts[1].getAccountBalance() +
                        job.accounts[2].getAccountBalance()  +
                        job.accounts[3].getAccountBalance()))) {
      approve = true;
    }
    cleanupFiles();
    return approve;
  }

  // Remove artificial files created
  private void cleanupFiles() {
    for (int i=0;i<job.COUNT;i++) {
      File a = new File(job.accounts[i].getAccountName());
      a.delete();
    }
  }
}
```

InputManager.java

This class manages all input for the banking application.

Java classes used include:

- `Button`
- `Dialog`
- `Frame`
- `Label`
- `Panel`
- `TextArea`
- `TextField`
- `FlowLayout`
- `Font`
- `GridBagConstraints`
- `GridBagLayout`

Java interfaces used include:

- `Runnable`
- `ActionListener`

Features of the sample code include:

- Use of AWT classes listed above
- Event handling—`ActionListener` interface
- Threads—`Runnable` interface
- Error handling—try, catch (`StringIndexOutOfBoundsException`, `NumberFormatException`)

Written by Peg MacPhail, IBM Technical Consultant

InputManager.java

```
package banking;

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

// InputManager extends Observable class so that the main processing routine
// can receive notice when the input for a loan application is complete.
// InputManager implements the Runnable interface so that the InputManager
// can run in a separate thread.
// InputManager implements the ActionListener interface so that the
// InputManager can register for and receive input action events.

public class InputManager extends Observable
                    implements Runnable, ActionListener {

  private TextArea resultsArea;     // Output area in display window
  private String name;              // Name of the person applying for loan
  private double amount;            // Amount of loan requested
  private Dialog mainDialog;
  private Panel buttonPanel;
  private TextField loaneeField, amountField;
  private Button okayButton, quitButton, cancelButton;
  private boolean done;
  private boolean disableInput;
  private final int resultsWidth = 40;

  // Constructor
  InputManager () {

     // Initialization of variables
     super();
     name = "";
     amount = 0;
     done = false;
     disableInput = false;

     // Create layout object
     GridBagConstraints gConsts = new GridBagConstraints();
     gConsts.fill = gConsts.NONE;
     gConsts.gridx = gConsts.CENTER;
     GridBagLayout layout = new GridBagLayout();

     Font theFont = new Font ("TimesRoman",Font.PLAIN,14);

     // Create input features
     Frame topFrame = new Frame();
     topFrame.setFont(theFont);
     topFrame.setForeground(Color.blue);

     mainDialog = new Dialog (topFrame);
     mainDialog.setLayout(layout);
     mainDialog.setResizable(true);
     mainDialog.setSize(350,550);
     mainDialog.setFont(theFont);
     mainDialog.setTitle("Loan Office");

     Label loaneeLabel = new Label("Please enter your name");
```

```
    mainDialog.add(loaneeLabel);
    layout.setConstraints(loaneeLabel,gConsts);
    loaneeField = new TextField(20);
    loaneeField.addActionListener(this);
    loaneeField.setEditable(true);
    mainDialog.add(loaneeField);
    layout.setConstraints(loaneeField,gConsts);

    Label amountLabel = new Label("Please enter the amount you want");
    mainDialog.add(amountLabel);
    layout.setConstraints(amountLabel,gConsts);
    amountField = new TextField(10);
    amountField.addActionListener(this);
    amountField.setEditable(true);
    mainDialog.add(amountField);
    layout.setConstraints(amountField,gConsts);

    buttonPanel = new Panel();

    FlowLayout buttonLayout = new FlowLayout();
    buttonPanel.setLayout(buttonLayout);

    okayButton = new Button("Okay");
    okayButton.addActionListener(this);
    buttonPanel.add(okayButton);

    quitButton = new Button("Quit");
    quitButton.addActionListener(this);
    buttonPanel.add(quitButton);

    cancelButton = new Button("Cancel");
    cancelButton.addActionListener(this);
    buttonPanel.add(cancelButton);

    mainDialog.add(buttonPanel);
    layout.setConstraints(buttonPanel,gConsts);

    Label resultsLabel = new Label("Results displayed");
    mainDialog.add(resultsLabel);
    gConsts.gridx = gConsts.CENTER;
    layout.setConstraints(resultsLabel,gConsts);
    resultsArea = new TextArea(20,resultsWidth);
    resultsArea.setEditable(true);
    mainDialog.add(resultsArea);
    layout.setConstraints(resultsArea,gConsts);
}

// This method is called when the InputManager thread is run
public void run() {
    mainDialog.show();
    buttonPanel.show();
    while (!done) {}
}

// This method is called to output information to the display window
public void appendText(String text) {
  if (text.length() <= resultsWidth) {
   resultsArea.append(text);
  }
  else {
```

```
      int i = text.lastIndexOf(" ");
      if (i <= resultsWidth) {
      try {
        resultsArea.append(text.substring(0,i) + "\n" +
           text.substring(i+1,text.length()) );
      }
      catch (StringIndexOutOfBoundsException e) {}
      }
      else {
      StringTokenizer stoke = new StringTokenizer(text);
      String outString = "";
      try {
        while (stoke.hasMoreElements()) {
          String holder = stoke.nextToken();
          if ( (holder.length()+outString.length()) < resultsWidth) {
            outString = outString + holder + " ";
          }
          else {
            if (outString.length()>0) {
            resultsArea.append(outString + "\n");
            }
            outString = holder + " ";
          }
        }
        if (outString.length()>0) {
          resultsArea.append(outString + "\n");
        }
      }
      catch (NoSuchElementException e) {}
      }
    }
  }

  // This method returns the name of the person, if any, requesting a loan
  public String getName() {
    return name;
  }

  // This method returns the amount requested, if any, for the loan
  public double getAmount() {
    return amount;
  }

  // This method is called when the okay button is pressed.  Notice that
  // this method is synchronized so that no one else can access the name
  // and the amount fields while they are being updated.
  public synchronized void okay() {
    if (!disableInput) {
      name = "";
      amount = 0;
      processFields();
      if (amount == 0) {
      resultsArea.append("No amount entered.\n");
      }
      if (name == "") {
      resultsArea.append("No name entered.\n");
      }
    }
  }
```

```
// This method is called when the quit button is pressed.  Notice that
// this method is synchronized so that no one else can access the name
// and the amount fields while quit is in process.
public synchronized void iQuit() {
  setChanged();                          // setChanged is an Observable
  notifyObservers("Quit");               // method which MUST be called
                              // in order for notifyObservers
                              // method to notify observers.
  System.exit(0);
}

// This method resets the input display and input variables. Notice that
// this method is synchronized so that no one else can access the name
// and the amount fields while reset is in process.
public synchronized void reset() {
  disableInput = false;
  name = "";
  amount = 0;
  amountField.setText("");
  loaneeField.setText("");
  loaneeField.requestFocus();
}

// This method is called when the cancel button is pressed. Notice that
// this method is synchronized so that no one else can access the name
// and the amount fields while cancel is in process.
public synchronized void iCancel() {
  if (!disableInput) {
    resultsArea.append("Cancel.  Fields cleared.\n");
    reset();
  }
  else {
    resultsArea.append("Not Cancelled, work in process.\n");
  }
}

// This method gets the data from the input fields - amount, name
public synchronized void processFields() {
  if (!disableInput) {
    name = loaneeField.getText();
    try {
    String amt = amountField.getText();
    if (amt.length() !=  0) {
      Double a = new Double(amt);
      amount = a.doubleValue();
    }
    }
    catch (NumberFormatException e) {
    amount = 0;
    resultsArea.append("Bad amount value.  Please re-enter amount.\n");
    amountField.selectAll();
    }
    if ((name.length() != 0) && (amount != 0) ) {
    appendText("\nLoan request for " + name +
               " for " + formatDollars(amount) + "\n");
    disableInput = true;
    setChanged();                    // setChanged is an Observable
    notifyObservers("Ready");        // method which MUST be called
                              // in order for notifyObservers
```

```
                                // method to notify observers.
      }
    }
    else {
      resultsArea.append("Work in process.\n");
    }
  }

  // This method is implemented for the ActionListener interface
  public void actionPerformed (ActionEvent e) {
    Object source = e.getSource();
    if ( (source == amountField) || (source == loaneeField) ||
       (source == okayButton) ) {
      processFields();
    }
    else if (source == quitButton) {
         iQuit();
       }
       else if (source == cancelButton) {
            iCancel();
            }
  }

  public String formatDollars(double d) {
    double f = Math.floor(d);
    Integer a = new Integer((int)f);
    int r = (int) (100*(d-f));
    if (r<0) r=-r;
    if (r<10) r=10*r;
    String b = "";
    if (r==0) {
      b = "00";
    }
    else {
      Integer x = new Integer(r);
      b = x.toString();
    }
    String out= "$" + a.toString() + "." + b;
    return out;
  }
}
```

Account.java

The `Account` class holds the information related to an individual bank account.

Sample Java classes found in other sample files include:

- `BankJob`
- `MyMath`
- `ProcessAccount`
- `Transaction`

Java classes used include:

- `File`
- `PrintStream`
- `Stack`
- `StreamTokenizer`
- `Vector`

Features of the sample code include:

- Reading and writing from a file
- Storing objects in a Vector and in a Stack
- Error handling—try, catch (`IOException`)

Written by Peg MacPhail, IBM Technical Consultant

Account.java

```
package banking;

import banking.BankJob;
import banking.MyMath;
import banking.ProcessAccount;
import banking.Transaction;
import java.util.*;
import java.io.*;
```

```
class Account {
  private String accountName;
  private String accountType;
  private BankJob job;
  private Vector transactions;             // Contains daily transactions
  private Stack processedTransactions;     // Contains transaction processed
  private double balance;                  // current balance
  private boolean amProcessing;

  Account(String  name, String type, BankJob j) {
    accountName = name;
    accountType = type;
    job = j;
    balance = 0;
    transactions = new Vector();
    processedTransactions = new Stack();
  }
  public synchronized void postTransaction(Transaction t) {
     if (amProcessing) {
       updateBalance(t.getAmount());
     }
     else {
       transactions.addElement(t);
     }
  }
  public void processTransactions(ProcessAccount work, int work_no ) {
    amProcessing = true;
    double subtotal = 0;
    int start = work.getStart();
    int stop = work.getStop();
    if ((stop < transactions.size()) && (start >=0)) {
      if ((stop-start) <= work_no) {
      for (int i=start;i<=stop;i++) {
        Transaction t = (Transaction) transactions.elementAt(i);
        subtotal += t.getAmount();
        processVerify(t.getAmount());
        addProcessed(t);
      }
      updateBalance(subtotal);
      }
      else {
      work.divideWork();
      }
    }
    else
       job.IM.appendText("Bad input to process transactions.\n start ="
           + start + " stop = " + stop + "\n");
  }
  private synchronized void updateBalance(double amount) {
    balance += amount;
  }
  private synchronized void addProcessed(Transaction t) {
    processedTransactions.push(t);
  }
  public synchronized void removeTransactions() {
    while(!processedTransactions.empty()) {
      transactions.removeElement(processedTransactions.pop());
    }
    if (transactions.isEmpty()){
      amProcessing = false;
```

```
    }
  }
  public synchronized void checkPointTransactions() {
    removeTransactions();
    amProcessing = false;
  }
  public String getAccountName() { return accountName; }
  public String getAccountType() { return accountType; }
  public double getAccountBalance() { return balance; }
  public int getTransCount() { return transactions.size(); }
  public void writeOut () {
    try {
      PrintStream ps =
      new PrintStream (new FileOutputStream(new File(accountName)));
      ps.println(accountName + " " + accountType + " " + balance + " ");
      while (!transactions.isEmpty()) {
      Transaction t = (Transaction) transactions.elementAt(0);
      t.writeOut(ps);
      transactions.removeElement(t);
      }
    }
    catch (IOException e) {
      job.IM.appendText("Unable to print to " + accountName);
    }
  }
  public int readIn ()  {
   int r=0;
   try {
     StreamTokenizer toke =
      new StreamTokenizer (new FileInputStream(new File(accountName)));
     r = toke.nextToken();
     if ® == toke.TT_WORD) {
       accountName = toke.sval;
       r = toke.nextToken();
       if (r == toke.TT_WORD) {
       accountType = toke.sval;
       r = toke.nextToken();
       if (r == toke.TT_NUMBER) {
         balance = toke.nval;
         Stack ts = new Stack();
         while (r != toke.TT_EOF) {
           Transaction t = new Transaction ();
           r = t.readIn(toke);
           if (r != toke.TT_EOF) {
             postTransaction(t);
           }
         }
       }
       }
    }
   }
   catch (FileNotFoundException e) {
    job.IM.appendText("File " + accountName + " not found\n");
   }
   catch (IOException e) {
    job.IM.appendText("Error reading in " + accountName + "\n");
   }
   readVerify();
   return r;
  }
```

```
    private void readVerify() {
      MyMath.verifyAccount(job.readBurden);
    }
    private void processVerify(double amt) {
      MyMath.verifyAmount(amt, job.readBurden);
    }
  }
```

Job.java

The `Job` class holds the "main" method for the banking application invoked with `java banking.Job` with two optional parameters for setting work burdens for the worker threads.

Java classes used include:

- `StreamTokenizer`

Features of the sample code include:

- Processing standard input to the main method
- Error handling—try, catch (`IOException`, `ArrayIndexOutOfBoundsException`)

Written by Peg MacPhail, IBM Technical Consultant

Job.java

```
package banking;

import banking.BankJob;
import java.io.*;

// This class is invoked from the command line "java banking.Job" with
// optional "burden" parameters.  The burden parameters are used to
// artificially load down the worker threads by looping through a math
// calculation "burden" times.
// The StreamTokenizer class is used to process the standard input.
public class Job {
  private static final int BURDEN = 1000;
  public static void main(String argv[])  {
    int readBurden = BURDEN;
    int processBurden = BURDEN;
    try {
      if (argv.length > 0) {
      StreamTokenizer toke =
           new StreamTokenizer(new StringReader(argv[0]));
      int r = toke.nextToken();
      if (r == toke.TT_NUMBER) {
```

```
          readBurden = (int) toke.nval;
          if (argv.length > 1) {
            toke =
             new StreamTokenizer(new StringReader(argv[1]));
            r = toke.nextToken();
            if (r == toke.TT_NUMBER) {
              processBurden = (int) toke.nval;
            }
          }
        }
        }
      }
      catch (IOException e) {}
      catch (ArrayIndexOutOfBoundsException e)  {}
      BankJob job = new BankJob (readBurden, processBurden);
      job.runJob();
    }
  }
```

MyMath.java

The MyMath class holds the methods for the banking math calculations. Some are dummy methods used to artificially load the worker threads.

Features of the sample code include:

- Static methods that make the MyMath class a utility; that is, this class does not have to be instantiated to be used

Java classes used include:

- Math

Written by Peg MacPhail, IBM Technical Consultant

MyMath.java

```
package banking ;

public class MyMath {

  // This is a dummy method that performs no useful work except to
  // artificially stress the worker threads
  public static int verifyAccount(int burden) {
     int i, r=0;
     double flier;

     for (i=0;i<burden;i++) {
       flier = Math.log((double)(i+2));
       flier = Math.log((double)(i+1));
     }
     return r;
   }

  // This is a dummy method that performs no useful work except to
  // artificially stress the worker threads
  public static int verifyAmount(double amount, int burden) {
    int i, r=0;
    double flier, a;
```

```
      for (i=0;i<burden;i++) {
        a = (amount - Math.floor(amount))/10;
        flier = Math.exp(7*Math.log(a))/a;
       }

      return r;
     }

    // This method will return a pseudo-random int less than or equal
    // to the value passed in the method call
    public static int getRandomNumber(int largest) {
      int result;
      double f, r;
      double multiplier = Math.max((double)largest,10);
      r = Math.random() * multiplier;
      for (int i=0;(i<5)&&(r<multiplier);i++) {
         r = r * multiplier;
      }
      r = Math.floor(r);
      while (r > largest) {
        f = Math.floor(r/(largest+1));
        r = r - (f*(largest+1));
      }
      result = (int) r;
      return result;
    }

}
```

ProcessAccount.java

This class processes the transactions for an account—getting transactions and changing the balance accordingly.

Other sample code Java classes used include:

- Account
- BankJob
- WorkItem

Features of the sample code include:

- Partial implementation of the Blumofe/Leiserson work-stealing algorithm. If the work is too large—larger than the number of transactions specified by the job WORKNO—then the work is divided into 2.

Written by Peg MacPhail, IBM Technical Consultant

ProcessAccount.java

```
package banking;

import banking.Account;
import banking.BankJob;
import banking.WorkItem;

class ProcessAccount extends WorkItem {
  private int start;
  private int stop;
  ProcessAccount(Worker wm, int i, int rc, BankJob j, int str, int stp) {
    super(wm, i, rc, "process", j);
    start = str;
    stop = stp;
    checkToPost();
  }
  public int getStart() {return start;}
  public int getStop() {return stop;}
  public void run () {
    job.accounts[index].processTransactions(this,job.WORKNUMBER);
```

```
    }
    public void divideWork() {
      int cusp = start+((stop-start)/2);
      ProcessAccount a =
         new ProcessAccount(workMaster, index, 0, job, start, cusp);
      a = new ProcessAccount(workMaster, index, 0, job, cusp+1, stop);
    }
  }
```

Transaction.java

This class holds the information for an account transaction.

Java classes used include:

- PrintStream
- StreamTokenizer

Features of the sample code include:

- File I/O
- Error handling—try, catch

Written by Peg MacPhail, IBM Technical Consultant

Transaction.java

```
package banking;

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

class Transaction {
  private double amount;
  private String accountName;
  Transaction(String name, double amt) {
    amount = amt;
    accountName = name;
  }
  Transaction() {
    amount = 0;
    accountName = “”;
  }
  public double getAmount() {return amount;}
  public String getAccountName() {return accountName;}
  public void writeOut(PrintStream ps)  {
   ps.println(accountName + “ “ + amount);
  }
  public int readIn(StreamTokenizer toke) throws IOException {
     int r = toke.nextToken();
```

```
      if ® == toke.TT_WORD) {
        accountName = toke.sval;
        r = toke.nextToken();
        if (r == toke.TT_NUMBER) {
        amount = toke.nval;
        }
      }
      return r;
    }
  }
```

Worker.java

This class contains the worker thread object.

Java classes used include:

- `Vector`

Java interfaces used include:

- `Runnable`

Other sample code Java classes used include:

- `BankJob`
- `MyMath`
- `WorkItem`

Features of the sample code include:

- Partial implementation of the Blumofe/Leiserson work-stealing algorithm
- Main thread method—run
- Error handling—try, catch (`IllegalMonitorStateException`, `NoSuchElementException`)
- Synchronization at object level

Written by Peg MacPhail, IBM Technical Consultant

Worker.java

```
package banking;

import banking.BankJob;
import banking.MyMath;
import banking.WorkItem;
import java.util.*;

class Worker implements Runnable {
  private BankJob job;
  private Vector workList;
  private boolean done;
```

```
    private int id;
    Worker(int i, BankJob j) {
      workList = new Vector();
      done = false;
      id = i;
      job = j;
     }
    // This method is called when the thread is started
    public void run() {
      while (!done) {
        WorkItem work = getBottomWork();
        if ( (work == null) && (!done) ) {
         work = stealWork();
        }
        if (work != null) {
        work.setWorkMaster(this);
        work.run();
        job.ender.decreaseRC();        // Job ender WorkItem has a
                                  // releaseCount = # of WorkItems
                                  // in all of the work lists
        }
        else {
        job.threads[id].yield();
        }
      }
    }
    private WorkItem stealWork() {
      int i = MyMath.getRandomNumber(job.NUMTHREADS-1);
      WorkItem  work = job.workers[i].getTopWork(id);
      return work;
    }
    public void postWork(WorkItem w) {
      job.ender.addRC();                  // Job ender WorkItem has a
                                  // releaseCount = # of WorkItems
                                  // in all of the work lists
      synchronized (workList) {
        workList.addElement(w);
      }
    }
    public WorkItem getTopWork(int i) {
      WorkItem w = null;
      try {
        synchronized (workList) {
        w = (WorkItem) workList.firstElement();
        workList.removeElement(w);
        }
      }
      catch (NoSuchElementException e) {}
      catch (IllegalMonitorStateException e) {}
      return(w);
    }
    public WorkItem getBottomWork() {
      WorkItem w = null;
      try {
        synchronized (workList) {
        w = (WorkItem) workList.lastElement();
        workList.removeElement(w);
        }
```

```
      }
      catch (NoSuchElementException e) {}
      catch (IllegalMonitorStateException e) {}
      return(w);
    }
    public void stop() {
      done = true;
    }
  }
```

WorkItem.java

This file contains an abstract base class—`WorkItem`. These `WorkItems` can be posted to the `Worker` queue, when the `releaseCount = 0`. Dependencies are added by increasing the `releaseCount` and removed by decreasing the `releaseCount`.

Other sample Java classes used from other files include:

- `Worker`

Features of the sample code include:

- Use of abstract base class
- Part of the implementation of the Blumofe/Leiserson work-stealing algorithm

Written by Peg MacPhail, IBM Technical Consultant

WorkItem.java

```
package banking;

import banking.BankJob;
import banking.Worker;
import java.util.*;

abstract class WorkItem {
  protected int index;                  // index to in job thread array
  protected Worker workMaster;          // worker object who owns this item
  protected BankJob job;                // overall job object
  private String workType;
  protected int releaseCount;           // count to release this item
  private boolean posted;               // posted to work queue when released
  WorkItem(Worker wm, int i, int rc, String wt, BankJob j) {
    workMaster = wm;
    index = i;
    releaseCount = rc;
    workType = wt;
    job = j;
    posted = false;
  }
```

```
    abstract public void run();
    public synchronized void setWorkMaster(Worker wm) {
      workMaster = wm;
    }
    public synchronized void decreaseRC () {
      releaseCount–;
      checkToPost();
    }
    public synchronized void checkToPost () {
      if ((releaseCount == 0) && (!posted)) {
        workMaster.postWork(this);
        posted = true;
      }
    }
    public synchronized void addRC () {
      releaseCount++;
    }
    public String getWorkType() {
      return workType;
    }
    public String getAccountType() {
      return job.acctTypes[index];
    }
  }
```