File Bank.java

```
public interface Bank
{
   /**
    * Add a customer to the bank.
    * @param threadNum The number of the customer being added.
    * @param maxDemand The maximum demand for this customer.
    */
        public void addCustomer(int threadNum, int[] maxDemand);

   /**
    * Outputs the available, allocation, max, and need matrices.
    */
        public void getState();

   /**
    * Make a request for resources.
    * @param threadNum The number of the customer being added.
    * @param maxDemand The request for this customer.
    * @return  true The request is granted.
    * @return  false The request is not granted.
         */
   public boolean requestResources(int threadNum, int[] request);

        /**
         * Release resources.
    * @param threadNum The number of the customer being added.
    * @param release The resources to be released.
         */
        public  void releaseResources(int threadNum, int[] release);
}
```

File infile.txt

```
7,5,3
3,2,2
9,0,2
2,2,2
4,3,3
```

File BankImpl.java

```
/**
 * The Bank
 */

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

public class BankImpl implements Bank
{
        private int n;                  // the number of threads in the system
        private int m;                  // the number of resources

        private int[] available;        // the amount available of eah resource
        private int[][] maximum;        // the maximum demand of each thread
        private int[][] allocation;     // the amount currently allocated to each thread
        private int[][] need;           // the remaining needs of each thread

        /**
         * Create a new bank with resources.
         */
        public BankImpl(int[] resources) {
                // m is the number of resources
                m = resources.length;
           n = Customer.COUNT;

                // initialize the resources array
                available = new int[m];
                System.arraycopy(resources,0,available,0,m);

           // create the array for storing the maximum demand by  each thread
           maximum = new int[Customer.COUNT][];
           allocation = new int[Customer.COUNT][];
           need = new int[Customer.COUNT][];
        }

      /**
       * This method is invoked by a thread when it enters the system. It records
       * its maximum demand with the bank.
       */
      public void addCustomer(int threadNum, int[] maxDemand) {
          maximum[threadNum] = new int[m];
          allocation[threadNum] = new int[m];
          need[threadNum] = new int[m];

          System.arraycopy(maxDemand, 0, maximum[threadNum], 0, maxDemand.length);
          System.arraycopy(maxDemand, 0, need[threadNum], 0, maxDemand.length);
      }

        /**
```

```
     * Outputs the state for each thread
     */

    public void getState() {
        System.out.print("Available = [");
        for (int i = 0; i < m-1; i++)
            System.out.print(available[i]+" ");
        System.out.println(available[m-1]+"]");
        System.out.print("\nAllocation = ");
        for (int i = 0; i < n; i++) {
            System.out.print("[");
            for (int j = 0; j < m-1; j++)
                System.out.print(allocation[i][j]+" ");
            System.out.print(allocation[i][m-1]+"]");
        }
        System.out.print("\nMax = ");
        for (int i = 0; i < n; i++) {
            System.out.print("[");
            for (int j = 0; j < m-1; j++)
                System.out.print(maximum[i][j]+" ");
            System.out.print(maximum[i][m-1]+"]");
        }
        System.out.print("\nNeed = ");
        for (int i = 0; i < n; i++) {
            System.out.print("[");
            for (int j = 0; j < m-1; j++)
                System.out.print(need[i][j]+" ");
            System.out.print(need[i][m-1]+"]");
        }

        System.out.println();
    }


    /**
     * Determines whether granting a request results in leaving
     * the system in a safe state or not.
     *
     * @return  true - the system is in a safe state.
     * @return  false - the system is NOT in a safe state.
     */
    private boolean isSafeState (int threadNum, int[] request) {
        System.out.print("\n Customer # " + threadNum + " requesting ");
        for (int i = 0; i < m; i++) System.out.print(request[i] + " ");

        System.out.print("Available = ");
        for (int i = 0; i < m; i++)
            System.out.print(available[i] + "  ");

        // first check if there are sufficient resources available
        for (int i = 0; i < m; i++)
            if (request[i] > available[i]) {
                System.err.println("INSUFFICIENT RESOURCES");
```

```
        return false;
    }

// ok, they're are. Now let's see if we can find an ordering of threads to finish
        boolean[] canFinish = new boolean[n];
        for (int i = 0; i < n; i++)
                canFinish[i] = false;

        // copy the available matrix to avail
        int[] avail = new int[m];
        System.arraycopy(available,0,avail,0,available.length);

        // Now decrement avail by the request.
// Temporarily adjust the value of need for this thread.
// Temporarily adjust the value of allocation for this thread.
        for (int i = 0; i < m; i++) {
         avail[i] -= request[i];
         need[threadNum][i] -= request[i];
         allocation[threadNum][i] += request[i];
}

        /**
         * Now try to find an ordering of threads so that
         * each thread can finish.
         */

        for (int i = 0; i < n; i++) {
                // first find a thread that can finish
                for (int j = 0; j < n; j++) {
                        if (!canFinish[j]) {
                                boolean temp = true;
                                for (int k = 0; k < m; k++) {
                                        if (need[j][k] > avail[k])
                                                temp = false;
                                }
                                if (temp) { // if this thread can finish
                                        canFinish[j] = true;
                                        for (int x = 0; x < m; x++)
                                                avail[x] += allocation[j][x];
                                }
                        }
                }
        }

// restore the value of need and allocation for this thread
for (int i = 0; i < m; i++) {
    need[threadNum][i] += request[i];
    allocation[threadNum][i] -= request[i];
}

// now go through the boolean array and see if all threads could complete
        boolean returnValue = true;
        for (int i = 0; i < n; i++)
```

```
                        if (!canFinish[i]) {
                                returnValue = false;
                                break;
                        }

                return returnValue;
        }

        /**
         * Make a request for resources. This is a blocking method that returns
         * only when the request can safely be satisfied.
         *
         * @return  true - the request is granted.
         * @return  false - the request is not granted.
       */

        public synchronized boolean requestResources(int threadNum, int[] request)  {
                if (!isSafeState(threadNum,request)) {
                 //System.out.println("Customer # " + threadNum + " is denied.");
                        return false;
           }

                // if it is safe, allocate the resources to thread threadNum
                for (int i = 0; i < m; i++) {
                 available[i] -= request[i];
                        allocation[threadNum][i] += request[i];
                        need[threadNum][i] = maximum[threadNum][i] - allocation[threadNum]
[i];
                }
           /*
           System.out.println("Customer # " + threadNum + " using resources.");
            System.out.print("Available = ");
           for (int i = 0; i < m; i++)
              System.out.print(available[i] + "  ");
            System.out.print("Allocated = [");
           for (int i = 0; i < m; i++)
              System.out.print(allocation[threadNum][i] + "  ");
              System.out.print("]");
           */
           return true;
        }


        /**
         * Make a request for resources. This is a blocking method that returns
         * only when the request can safely be satisfied.
         *
         * @return  true - the request is granted.
         * @return  false - the request is not granted.

        public synchronized void requestResources(int threadNum, int[] request) throws
InterruptedException {
                while (!isSafeState(threadNum,request))
```

```
                        wait();

                    // if it is safe, allocate the resources to thread threadNum
                    for (int i = 0; i < m; i++) {
                     available[i] -= request[i];
                            allocation[threadNum][i] += request[i];
                            need[threadNum][i] = maximum[threadNum][i] - allocation[threadNum]
[i];
                    }
                System.out.println("Customer # " + threadNum + " using resources.");
                 System.out.print("Available = ");
                for (int i = 0; i < m; i++)
                   System.out.print(available[i] + "  ");
              }
             */

            /**
             * Release resources
                *
                * @param int[] release - the resources to be released.
             */
            public  synchronized void releaseResources(int threadNum, int[] release)  {
                System.out.print("\n Customer # " + threadNum + " releasing ");
                for (int i = 0; i < m; i++) System.out.print(release[i] + " ");

                    for (int i = 0; i < m; i++) {
                            available[i] += release[i];
                            allocation[threadNum][i] -= release[i];
                            need[threadNum][i] = maximum[threadNum][i] + allocation[threadNum]
[i];
                    }

                System.out.print("Available = ");
                for (int i = 0; i < m; i++)
                   System.out.print(available[i] + "  ");

                System.out.print("Allocated = [");
                for (int i = 0; i < m; i++)
                   System.out.print(allocation[threadNum][i] + "  ");
                   System.out.print("]");

                    // there may be some threads that can now proceed
                    //notifyAll();
            }

}
```

File Factory.java

```
/**
 * A factory class that creates (1) the bank and (2) each customer at the bank.
 *
 * Usage:
 *  java Factory <one or more resources>
 *
 * I.e.
 *  java Factory 10 5 7
 */

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

public class Factory
{
   public static void main(String[] args) {
                int numOfResources = args.length;
       int[] resources = new int[numOfResources];
       for (int i = 0; i < numOfResources; i++)
          resources[i] = Integer.parseInt(args[i].trim());

       Bank theBank = new BankImpl(resources);
                 int[] maxDemand = new int[numOfResources];

                 // the customers
                 Thread[] workers = new Thread[Customer.COUNT];

       // read initial values for maximum array
       String line;
       try {
          BufferedReader inFile = new BufferedReader(new FileReader("infile.txt"));

          int threadNum = 0;
                        int resourceNum = 0;

                        for (int i = 0; i < Customer.COUNT; i++) {
                            line = inFile.readLine();
                            StringTokenizer tokens = new StringTokenizer(line,",");

                            while (tokens.hasMoreTokens()) {
                                int amt =
Integer.parseInt(tokens.nextToken().trim());
                                maxDemand[resourceNum++] = amt;
                            }
                            workers[threadNum] = new Thread(new Customer(threadNum,
maxDemand, theBank));
                            theBank.addCustomer(threadNum,maxDemand);
                            //theBank.getCustomer(threadNum);
                            ++threadNum;
                            resourceNum = 0;
                        }
                 }
       catch (FileNotFoundException fnfe) {
          throw new Error("Unable to find file \"infile.txt\"");
       }
       catch (IOException ioe) {
          throw new Error("Error processing \"infile.txt\"");
       }

                 // start all the customers
                 System.out.println("FACTORY: created threads");
```

```
                for (int i = 0; i < Customer.COUNT; i++)
                    workers[i].start();

                System.out.println("FACTORY: started threads");

                /**
                try { Thread.sleep(5000); } catch (InterruptedException ie) { }
                System.out.println("FACTORY: interrupting threads");
                for (int i = 0; i < Customer.COUNT; i++)
                    workers[i].interrupt();
                */
    }
}
```

File SleepUtilities.java

```
/**
 * utilities for causing a thread to sleep.
 * Note, we should be handling interrupted exceptions
 * but choose not to do so for code clarity.
 */

public class SleepUtilities
{
    /**
     * Nap between zero and NAP_TIME seconds.
     */
    public static void nap() throws InterruptedException {
       nap(NAP_TIME);
    }

    /**
     * Nap between zero and duration seconds.
     */
    public static void nap(int duration) throws InterruptedException {
           int sleeptime = (int) (NAP_TIME * Math.random() );
           try { Thread.sleep(sleeptime*1000); }
           catch (InterruptedException e) { throw e;}
    }

    private static final int NAP_TIME = 5;
}
```

File TestHarness.java

```
/**
 * A test harness for the bankers algorithm.
 *
 * Usage:
 *  java TestHarness <one or more resources>
 *
 * I.e.
 *  java TestHarness <input file> 10 5 7
 *
 * Once this is entered, the user enters "*" to output the state of the bank.
 */

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

public class TestHarness
{
   public static void main(String[] args) throws java.io.IOException {
      if (args.length < 1) {
         System.err.println("Usage java TestHarness <input file> <R1> <R2> ...");
         System.exit(-1);
      }

      // get the name of the input file
      String inputFile = args[0];

      // now get the resources
               int numOfResources = args.length-1;

      // the initial number of resources
      int[] initialResources= new int[numOfResources];

      // the resources involved in the transaction
      int[] resources= new int[numOfResources];
      for (int i = 0; i < numOfResources; i++)
         initialResources[i] = Integer.parseInt(args[i+1].trim());

      // create the bank
      Bank theBank = new BankImpl(initialResources);
               int[] maxDemand = new int[numOfResources];

      // read initial values for maximum array
      String line;
      try {
         BufferedReader inFile = new BufferedReader(new FileReader(inputFile));

         int threadNum = 0;
                        int resourceNum = 0;

                        for (int i = 0; i < Customer.COUNT; i++) {
                            line = inFile.readLine();
                            StringTokenizer tokens = new StringTokenizer(line,",");

                            while (tokens.hasMoreTokens()) {
                                int amt =
Integer.parseInt(tokens.nextToken().trim());
                                maxDemand[resourceNum++] = amt;
                            }
                            theBank.addCustomer(threadNum,maxDemand);
                            ++threadNum;
                            resourceNum = 0;
```

```
                    }
                }
        catch (FileNotFoundException fnfe) {
           throw new Error("Unable to find file " + inputFile);
        }
        catch (IOException ioe) {
           throw new Error("Error processing " + inputFile);
        }

                // now loop reading requests

                BufferedReader cl = new BufferedReader(new
InputStreamReader(System.in));
                int[] requests = new int[numOfResources];
                String requestLine;

                while ( (requestLine = cl.readLine()) != null) {
             if (requestLine.equals(""))
           continue;

                    if (requestLine.equals("*"))
           // output the state
                        theBank.getState();
                    else {
                        // we know that we are reading N items on the command line
                        // [RQ || RL] <customer number> <resource #1> <#2> <#3>
                        StringTokenizer tokens = new StringTokenizer(requestLine);

           // get transaction type - request (RQ) or release (RL)
           String trans = tokens.nextToken().trim();

           // get the customer number making the tranaction
                        int custNum = Integer.parseInt(tokens.nextToken().trim());

           // get the resources involved in the transaction
           for (int i = 0; i < numOfResources; i++) {
                            resources[i] =
Integer.parseInt(tokens.nextToken().trim());
                            System.out.println("*"+resources[i]+"*");
           }

           // now check the transaction type
           if (trans.equals("RQ")) {  // request
                       if (theBank.requestResources(custNum,resources))
                              System.out.println("Approved");
                       else
                              System.out.println("Denied");
           }
           else if (trans.equals("RL")) // release
              theBank.releaseResources(custNum, resources);
           else // illegal request
              System.err.println("Must be either 'RQ' or 'RL'");
                    }
                }
         }
}
```