Intermediate Software Development – MiniProject 3

Project Description:

The goal of this project is to demonstrate the producer-consumer parallel pattern. In this project we need to design four classes Product, ProductMessage, ProductProducer and ProductConsumer. Product class should contain fields that encapsulate the data from the PRODUCT_data.txt file. ProductMessage class should contain fields that encapsulate a Product object, the current timestamp and a random region identifier of a ProductConsumer. ProductProducer should produces a Product for consumption by the ProductConsumers. It should randomly select a product from the product list() and use that random product for populating ProductMessage with current timestamp, and random region Identifier. The ProductProducer should then pushes this message into its internal queue. ProductConsumer class should consume respective ProductMessage objects from ProductProducer based on the region ID and should maintain an internal list of its collected products. The simulation of the project should start from creating one producer and four consumer objects. At the end of the simulation the we need to show the total time taken for simulation, write all the consumed products into a file based on region, also display the all consumed products on the console. We need to end the simulation by pressing the enter key.

Installation, Compile and Run Time requirements:

1. NetBeans IDE 7.2.1
2. Java 1.7.0_11, Java Hotspot(TM) 64-Bit Server VM 23.6-b04
3. Windows 7 version 6.1 running on amd64

Insights, Expected results, and Challenges:

Insights:

This project helped me get experience on the concepts of Concurrency(Threads), Producer-Consumer parallel pattern,Basic I/o streams(File I/O), Collections (List),

While reading the data from given txt file we used BufferReader, and again while writing data to file we used File writer thus implementing one of the concept of Basic I/O operations. Collections are mainly used to group multiple elements into single object. All the data read from file are stored in the List of Product which has the fields to store data of read items. The same list is used for further computation.

Producer consumer pattern is started by Producer placing items of work on the queue for later processing instead of dealing with them the moment they are identified. The Consumer is then free to remove the work item based on the requirement from the queue for processing at any time in the future. Producer and Consumer contains a shared queue in which producer places the work created and consumer removes the work from that queue. In this project, the producer produces the ProductMessage based on the region ID and place it into the queue and whenever the consumer wants the product based on respective region ID removes the ProductMessage and use it.

If the required ProductMessage not present in the queue, consumer had to wait until the producer produce it and place it in queue or when queue is full then producer has to wait for consumer to read the ProductMessage from queue. At that point of time concurrency comes into picture. Errors due to this concurrency are handled using the "synchronized" keyword on shared methods. Both producer and consumer are implemented using thread concepts that is Runnable.

The packages used in this project are:

1. domain
2. domain.util
3. driver

1) **domain** package consists of the following classes:

I. Product
II. ProductMessage
III. ProductProducer
IV. ProductConsumer

I. Product class:

The fields of this class are used to store information about the product. This class does the following:

- □ Import the required packages.
- □ Declaration of variables to store info about the product.
- □ Providing No-arguments constructor.
- □ Providing Full-arguments constructor.
- □ Providing accessors and mutators for the variables declared.
- □ Providing toString() method.

II. ProductMessage class:

This class is used to encapsulate a Product object, the current timestamp and a random region identifier which is further used by ProductProducer and ProductConsumer. It does the following operations:

- □ Import the required package
- □ Declaration of variables to store Product object, timestamp and random region identifier.
- □ Providing No-arguments constructor.
- □ Providing Full-arguments constructor.
- □ Providing accessors and mutators for the variables declared.
- □ Providing toString() method.

III. ProductProducer class:

This class considered as the producer class and it prepares a ProductMessage by populating a ProductMessage with a random Product, the current timestamp and the ProductConsumer's region which subsequently pushes this message into its internal queue. It does the following operations:

- □ Import the required packages.

- □ Declaration of variables for length of queue, List name to store messages
- □ Implements Runnable.
- □ Function which randomly selects a region ID for product distribution.
- □ Prepares a ProductMessage by populating the message with random product object, current timestamp and region ID.
- □ Pushes the ProductMessage object in its internal queue .
- □ Has two synchronized function for pushing and getting the message from the queue(List).

IV. ProductConsumer class :

This class consumes only their respective ProductMessage objects produced by the ProductProducer from queue and maintains its own internal list of its collected products. This class does the following operations:

- □ Import the required packages.
- □ Declaration of variables to store region identifier, List variable to store consumed messages.
- □ Implements Runnable.
- □ After consuming the message, the details of the product are displayed.

2) **domain.util** package consist of the following class:

I. Utilities :

This class is used to implement methods which will called from driver class. This class helps in minimizig main() methods and understanding of the main class in a better and easy way. This class has methods to read txt file and to print and write consumed data on console and file respectively. The functionalities of all the methods are as described below:

readTextDataIntoArray () :

In this method, I create the instance of Bufferreader using which I read data from txt file format it as Product object and store that into an List.

exitAndPrintRequiredData(…….):

This method is called when user stops the application by hitting return key. It writes consumed Product data object to a file with respect to its regions and prints all consumed Product data object on console.

printAllConsumedObjects():

This method prints all consumed data objects on console. This method is called from *exitAndPrintRequiredData().*

writeConsumedDataToFile():

This method writes data to a file based on regions. This method is called from *exitAndPrintRequiredData().*

3) **driver** package consist of the following class:

I. MiniProject3 :

This is the main class from where the project execution starts and performs the following operations:

- □ Reads data from txt file convert it into product object and stores in the arraylist. This arraylist is further used by producer and consumers.

- Creates a Producer object and four Consumer objects. Producers will produce the require message and pushes in the queue and consumers consumes their respective message when it is ready in the queue.
- Implements the feature to stop the simulation whenever return key is pressed.
- Before exiting it prints consumed data by consumers to a file named PconsumerBasedOnRegion.txt based on respective regions, prints all consumed data by consumers onto console, computes and prints the simulation time on to console.

Expected Results:

Expected Results	*Test Result*
1. Should begin the simulation by creating and starting the ProductProducer and ProductConsumer objects.	Pass
2. Should provide a capability to terminate the simulation by a single keystroke.	Pass
3. Should display real-time queue change status in the ProductProducer .	Pass
4. Should display real-time consumption data per ProductConsumer.	Pass
5. Should write all Product objects per ProductConsumer to a file based on region.[If no products are produced to a region then nothing is displayed nor a single file w.r.t region is generated.]	Pass
6. Should display all Product objects per ProductConsumer on completion.	Pass
7. Should display the total elapsed time of the simulation.	Pass
8. Displays total no. of products produces and totals no of product consumed [Not Mandatory as per the requirement]	Pass

Note: To write all Product objects per ProductConsumer to a file based on region I was confused to write all data to single file or to create separate files based on region and write into it. So, I have implemented both ways to do that. If no products are produced to a region then nothing is displayed nor single file w.r.t region generated.

Challenges Faced:

While coding the project, there were no major challenges as such. But did encounter minor hiccups:

1. While writing data to file as we need to clear the file contents whenever simulation repeats. So, I decided to delete all output files before starting the simulation and when simulation ends fresh copy of files with data will be created.
2. Also, to end the simulation whenever return key is pressed I was confused to between scanner and system.in.read(). Based prof inputs and videos I decided to use sytem.in.read().
3. While return key is pressed the concurrentmodificationexception was occurring while displaying and writing data to console and file respectively. This was due to because the thread would be running at background and modifying the list which will be used to display and write the data. So, to avoid the thread to modify list I used the Boolean flag in both ProductProducer and ProductConsumer which will be used to stop thread gracefully before displaying data.

Screenshots:

1. Product Class

Snapshot 1-1:

```
//This class encapsulates the fields from the PRODUCT_data.txt file
public class Product {
    //Fields for the contents of Product_data.txt file
    private int PRODUCT_ID;
    private int MANUFACTURER_ID;
    private String PRODUCT_CODE;
    private float PURCHASE_COST;
    private int QUANTITY_ON_HAND;
    private float MARKUP;
    private String AVAILABLE;
    private String DESCRIPTION;

    //No-arg Constructor
    public Product() {
    }

    //Full-arg constructor.
    public Product(int PRODUCT_ID, int MANUFACTURER_ID, String PRODUCT_CODE,
                   float PURCHASE_COST,int QUANTITY_ON_HAND, float MARKUP,
                   String AVAILABLE, String DESCRIPTION) {
        this.PRODUCT_ID = PRODUCT_ID;
        this.MANUFACTURER_ID = MANUFACTURER_ID;
        this.PRODUCT_CODE = PRODUCT_CODE;
        this.PURCHASE_COST = PURCHASE_COST;
        this.QUANTITY_ON_HAND = QUANTITY_ON_HAND;
        this.MARKUP = MARKUP;
        this.AVAILABLE = AVAILABLE;
        this.DESCRIPTION = DESCRIPTION;
    }
```

Product class which encapsulates the fields present in txt file. Constructors with no-arg and full-arg

Snapshot 1-2:

```
//retrieve product id.
public int getPRODUCT_ID() {
    return PRODUCT_ID;
}
//store product id.
public void setPRODUCT_ID(int PRODUCT_ID) {
    this.PRODUCT_ID = PRODUCT_ID;
}
//retrieve manufacture id.
public int getMANUFACTURER_ID() {
    return MANUFACTURER_ID;
}
//store manufacture id.
public void setMANUFACTURER_ID(int MANUFACTURER_ID) {
    this.MANUFACTURER_ID = MANUFACTURER_ID;
}
//retrieve product code.
public String getPRODUCT_CODE() {
    return PRODUCT_CODE;
}
//store product code.
public void setPRODUCT_CODE(String PRODUCT_CODE) {
    this.PRODUCT_CODE = PRODUCT_CODE;
}
//retrieve purchase cost.
public float getPURCHASE_COST() {
    return PURCHASE_COST;
}
```

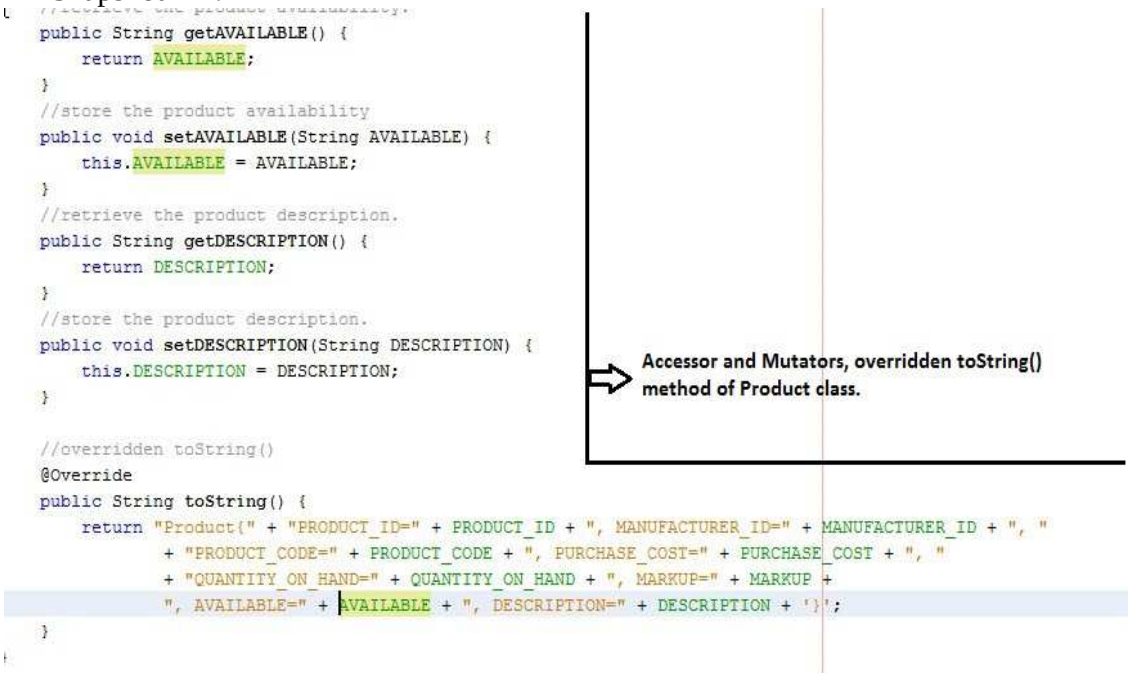
Accessors and Mutators for the fields of product class

Snapshot 1-3:

```
//store purchase cost.
public void setPURCHASE_COST(float PURCHASE_COST) {
    this.PURCHASE_COST = PURCHASE_COST;
}
//retrieve quantity on hand.
public int getQUANTITY_ON_HAND() {
    return QUANTITY_ON_HAND;
}
//store quantity on hand.
public void setQUANTITY_ON_HAND(int QUANTITY_ON_HAND) {
    this.QUANTITY_ON_HAND = QUANTITY_ON_HAND;
}
//retrieve mark up
public float getMARKUP() {
    return MARKUP;
}
//store mark up.
public void setMARKUP(float MARKUP) {
    this.MARKUP = MARKUP;
}
//retrieve the product availability.
public String getAVAILABLE() {
    return AVAILABLE;
}
//store the product availability
public void setAVAILABLE(String AVAILABLE) {
    this.AVAILABLE = AVAILABLE;
}
```

Accessors and Mutators for the fields of Product class

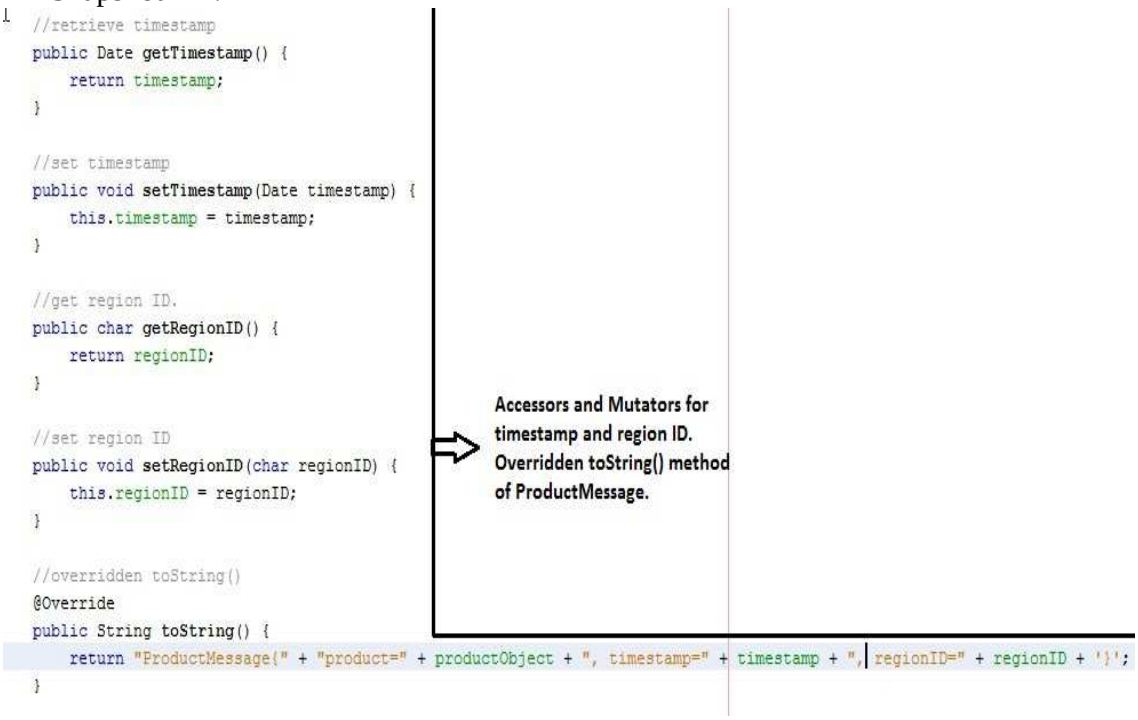
Snapshot 1-4:



2. ProductMessage class:

Snapshot 2-1:

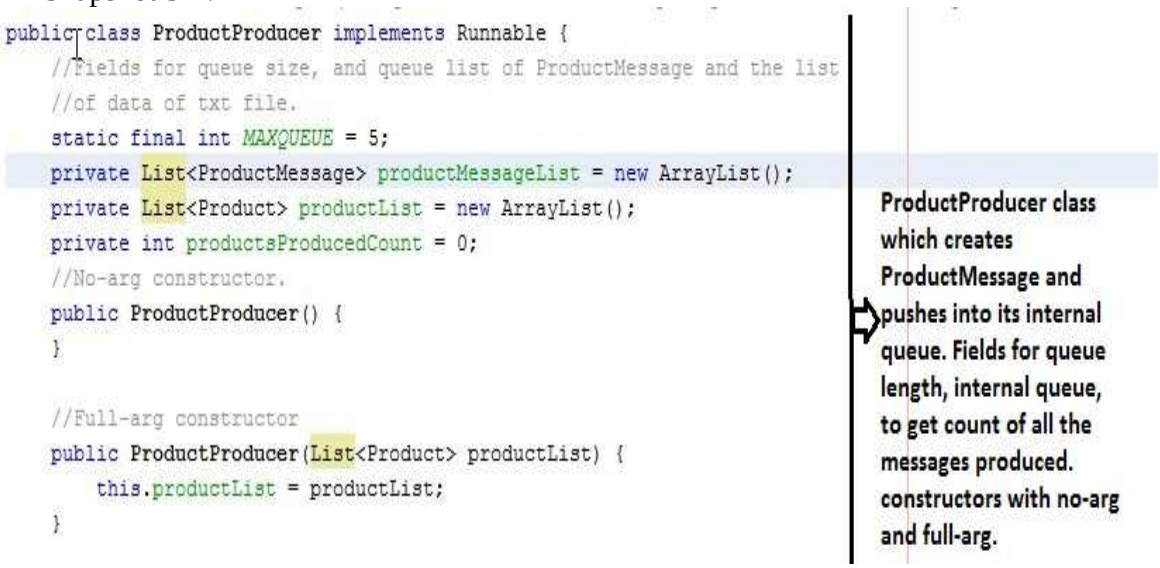
Snapshot 2-2:




```
//retrieve timestamp
public Date getTimestamp() {
    return timestamp;
}

//set timestamp
public void setTimestamp(Date timestamp) {
    this.timestamp = timestamp;
}

//get region ID.
public char getRegionID() {
    return regionID;
}

//set region ID
public void setRegionID(char regionID) {
    this.regionID = regionID;
}

//overridden toString()
@Override
public String toString() {
    return "ProductMessage{" + "product=" + productObject + ", timestamp=" + timestamp + ", regionID=" + regionID + '}';
}
```

3. ProductProducer class :

Snapshot 3-1:




```
public class ProductProducer implements Runnable {
    //Fields for queue size, and queue list of ProductMessage and the list
    //of data of txt file.
    static final int MAXQUEUE = 5;
    private List<ProductMessage> productMessageList = new ArrayList();
    private List<Product> productList = new ArrayList();
    private int productsProducedCount = 0;
    //No-arg constructor.
    public ProductProducer() {
    }

    //Full-arg constructor
    public ProductProducer(List<Product> productList) {
        this.productList = productList;
    }
```

Snapshot 3-2:

```
@Override
public void run() {
    while (true) {
        //get random product form the list which holds product_data.
        Product randomProduct = generateRandomProduct(productList);
        //Using randomproduct create producer message which is of ProductMessage
        //encapsulating randomproduct,timestamp, random region ID.
        ProductMessage pMessage = createProductMessage(randomProduct);
        //After creating producer message push the message into queue.
        putMessage(pMessage);
        try {
            //Make thread sleep.
            Thread.sleep(600);
        } catch (InterruptedException e) {
        }
    }
}

//Method to create random product from list which contains products data.
private static Product generateRandomProduct(List<Product> productList) {
    Product randomProduct = null;
    //creates a new random number generator.
    Random rnd = new Random();
    //Return the next pseudorandom, uniformly distributed int value from
    //this random number generators sequence.
    int rndIndex = rnd.nextInt(productList.size());
    return randomProduct = productList.get(rndIndex);
}
```

Method to create random ProductMessage by retrieving random product from List of products and invokes method to push the produced message into its internal queue.

Snapshot 3-3:

```
private static ProductMessage createProductMessage(Product randomProduct) {
    ProductMessage productMessage = null;
    //array of region IDs.
    char[] regionIds = {'N', 'S', 'E', 'W'};
    //creates a new random number generator.
    Random rnd = new Random();
    //Return the next pseudorandom, uniformly distributed int value from
    //this random number generators sequence.
    int rndIndex = rnd.nextInt(4);
    return productMessage = new ProductMessage(randomProduct, new Date(), regionIds[rndIndex]);
}

private synchronized void putMessage(ProductMessage pMessage) {
    //check queue size : if reached greater or equal to MAXQUEUE
    //wait until data to read from queue.
    while (productMessageList.size() >= MAXQUEUE) {
        try {
            wait();
        } catch (InterruptedException e) {
        }
    }
    //Otherwise add generated ProductMessage into queue.
    productMessageList.add(pMessage);
    productsProducedCount++;
    System.out.println("Product Produced for " + pMessage.getRegionID() + " region.\nQueue has " +
                       productMessageList.size() + " product(s).");
    notify();
}
```

Method to get random region ID which is used to create ProductMessage of random Product Object, timestamp, and region ID. Also, method to push creates ProductMessage to its internal queue by checking queue length and prints the queue status on console.

Snapshot 3-4:

```
//retrieve produced ProductMessages from queue(List).
public int getProductProducedCount() {
    return productsProducedCount;
}

//method called by consumer : To get respective message from queue(List)
//based on regionID.
public synchronized ProductMessage getMessage(char regionID) {
    ProductMessage pmessage = null;
    //If queue(List) is empty, wait until data is pused into list.
    while (productMessageList.size() == 0) {
        try {
            notify();
            wait();
        } catch (InterruptedException e) {
        }
    }
    //Read and remove ProductMessage from queue(List)
    //if its equals to respective regionID.
    if (regionID == productMessageList.get(0).getRegionID()) {
        pmessage = ((ProductMessage) productMessageList.remove(0));
    }
    notify();
    return pmessage;
}
```

Method to get count of produces messages, and to retrieve message from list based on region ID by checking whether the list is empty or not.

4. ProductConsumer class:

Snapshot 4-1:

```
public class ProductConsumer implements Runnable {
    //to encapsulate region,store consumed products and producer data
    private char regionID;
    List<ProductMessage> consumedProducts = new ArrayList<ProductMessage>();
    private ProductProducer producer;
    private long sleepInterval;
    private boolean stop = true;

    //No-arg constructors.
    public ProductConsumer() {
    }

    //full arg constructors
    public ProductConsumer(char regionID, ProductProducer producer, long sleepInterval) {
        this.regionID = regionID;
        this.producer = producer;
        this.sleepInterval = sleepInterval;
    }
```

ProductConsumer class fields and constructors with no-arg and full-arg.

Snapshot 4-2:

```
@Override
public void run() {
    while (stop) {
        //get message from producer if exists, based on the respective region.
        ProductMessage prodMessage = producer.getMessage(regionID);
        //on successful retrieveing of message
        if (prodMessage != null) {
            //Consumer consumes data and stores into consumedProducts List of ProductMessage class.
            consumedProducts.add(prodMessage);
            //To display the consumption data per ProductConsumer consumed.
            System.out.println("*********************************************************");
            System.out.println(prodMessage.getRegionID() + " region Consumer - Got Product");
            System.out.println("Product info of " + prodMessage.getRegionID()+":\n");
            System.out.println(prodMessage.toString());
            System.out.println("*********************************************************");
            try {
                Thread.sleep(sleepInterval);
            } catch (InterruptedException ex) {
                Logger.getLogger(ProductConsumer.class.getName()).log(Level.SEVERE, null, ex);
            }
        }
    }
}
```

ProductConsumer class which reads messages from queue based on its region ID and prints the consumed message info on the console.

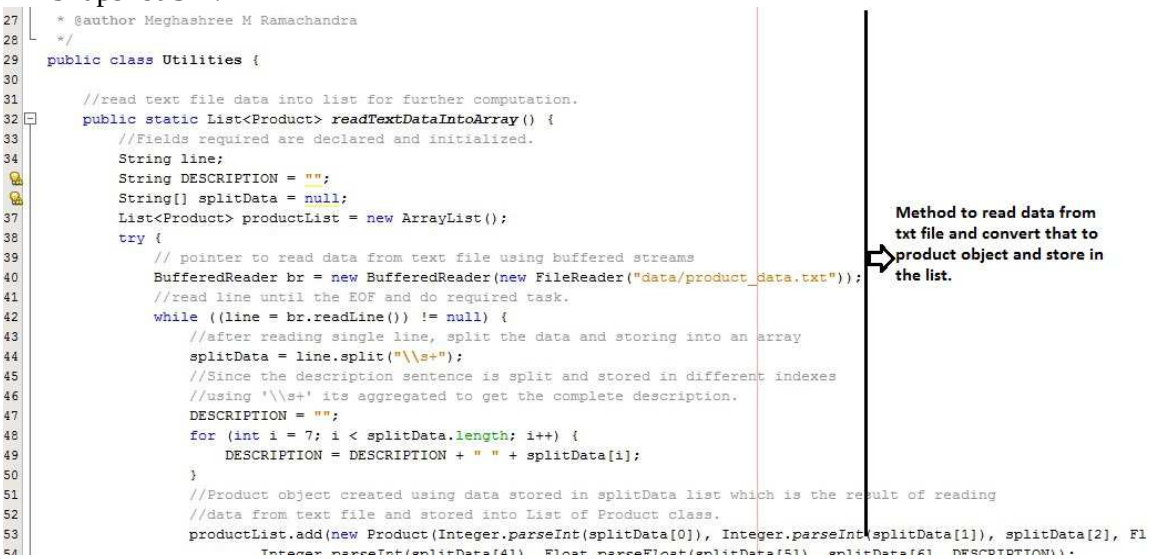
Snapshot 4-3:

```
//to retrieve the consumed products.
public List<ProductMessage> getConsumedProducts() {
    return consumedProducts;
}

public int getConsumedProductsListSize(){
    return consumedProducts.size();
}
```

code to get consumed products list and no of products consumed.

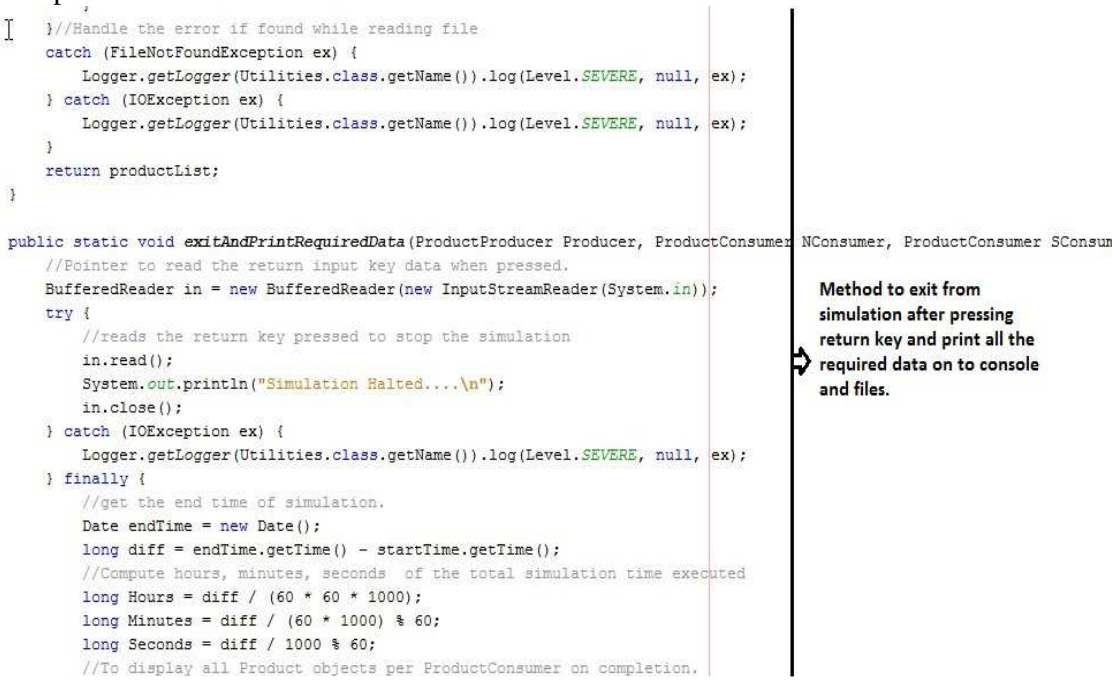
5. Utilities :

Snapshot 5-1:




```
 * @author Meghashree M Ramachandra
 */
public class Utilities {

    //read text file data into list for further computation.
    public static List<Product> readTextDataIntoArray() {
        //Fields required are declared and initialized.
        String line;
        String DESCRIPTION = "";
        String[] splitData = null;
        List<Product> productList = new ArrayList();
        try {
            // pointer to read data from text file using buffered streams
            BufferedReader br = new BufferedReader(new FileReader("data/product_data.txt"));
            //read line until the EOF and do required task.
            while ((line = br.readLine()) != null) {
                //after reading single line, split the data and storing into an array
                splitData = line.split("\\s+");
                //Since the description sentence is split and stored in different indexes
                //using '\\s+' its aggregated to get the complete description.
                DESCRIPTION = "";
                for (int i = 7; i < splitData.length; i++) {
                    DESCRIPTION = DESCRIPTION + " " + splitData[i];
                }
                //Product object created using data stored in splitData list which is the result of reading
                //data from text file and stored into List of Product class.
                productList.add(new Product(Integer.parseInt(splitData[0]), Integer.parseInt(splitData[1]), splitData[2], Fl
                        Integer.parseInt(splitData[4]), Float.parseFloat(splitData[5]), splitData[6], DESCRIPTION));
```

Snapshot 5-2:




```
    }//Handle the error if found while reading file
    catch (FileNotFoundException ex) {
        Logger.getLogger(Utilities.class.getName()).log(Level.SEVERE, null, ex);
    } catch (IOException ex) {
        Logger.getLogger(Utilities.class.getName()).log(Level.SEVERE, null, ex);
    }
    return productList;
}

public static void exitAndPrintRequiredData(ProductProducer Producer, ProductConsumer NConsumer, ProductConsumer SConsum
    //Pointer to read the return input key data when pressed.
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    try {
        //reads the return key pressed to stop the simulation
        in.read();
        System.out.println("Simulation Halted....\n");
        in.close();
    } catch (IOException ex) {
        Logger.getLogger(Utilities.class.getName()).log(Level.SEVERE, null, ex);
    } finally {
        //get the end time of simulation.
        Date endTime = new Date();
        long diff = endTime.getTime() - startTime.getTime();
        //Compute hours, minutes, seconds  of the total simulation time executed
        long Hours = diff / (60 * 60 * 1000);
        long Minutes = diff / (60 * 1000) % 60;
        long Seconds = diff / 1000 % 60;
        //To display all Product objects per ProductConsumer on completion.
```

Snapshot 5-3:

```
        //To display all Product objects per ProductConsumer on completion.
        System.out.println("Displaying all product objects per product consumer\n");
        printAllConsumedObjects(NConsumer.getConsumedProducts());
        printAllConsumedObjects(SConsumer.getConsumedProducts());
        printAllConsumedObjects(EConsumer.getConsumedProducts());
        printAllConsumedObjects(WConsumer.getConsumedProducts());
         //To write all Product objects per ProductConsumer consumed to a file based on region.
        System.out.println("Writing respective product objects per product consumer based on region to a file.\n");
        writeConsumedDataToFile('N', NConsumer.getConsumedProducts());
        writeConsumedDataToFile('S', SConsumer.getConsumedProducts());
        writeConsumedDataToFile('E', EConsumer.getConsumedProducts());
        writeConsumedDataToFile('W', WConsumer.getConsumedProducts());
        //To display the total elapsedtime of the simulation.
        System.out.println("Total Elapsed Time :   " + Hours + "Hours " + Minutes + "Minutes " + Seconds + "Seconds.\n");
        System.out.println("Total Products Produced : " + Producer.getProductProducedCount());
        System.out.println("Total Products Consumed By All Regions(N, S, E, W): " + (NConsumer.getConsumedProductsListSiz
                + EConsumer.getConsumedProductsListSize()+ WConsumer.getConsumedProductsListSize()));
    }
    System.exit(0);
}
```

Method to exit and print data continued

Snapshot 5-4:

```
//displays all consumed data by consumers(ie. every region)
private static void printAllConsumedObjects(List<ProductMessage> consumedProducts) {
    //check if there is products or not
    if (consumedProducts != null) {
        for (ProductMessage pm : consumedProducts) {
            System.out.println(pm);
        }
    }
}

public static void writeConsumedDataToFile(char dir, List<ProductMessage> pm) {
    try {
        //Pointer to create file.
        File file1 = new File("output/PConsumerBasedOnRegions.txt");
        File file2 = new File("output/" + dir + ".csv");
        //if file not exists, then create it
        if (!file1.exists()) {
            file1.createNewFile();
        }
        //if file doesnt exists, then create it
        if (!file2.exists()) {
            file2.createNewFile();
        }
```

Methods that print all consumed data to console and to a file based on regions

Snapshot 5-5:

```
            //Create a buffered file stream
            FileWriter fstream1 = new FileWriter("output/PConsumerBasedOnRegions.txt", true);
            BufferedWriter bufferWriter1 = new BufferedWriter(fstream1);
            FileWriter fstream2 = new FileWriter("output/" + dir + ".csv", true);
            BufferedWriter bufferWriter2 = new BufferedWriter(fstream2);

            //Add regions header into bufferwriter1
            bufferWriter1.write("Products for " + dir + " region : \n");
            bufferWriter2.write("Products for " + dir + " region : \n");
            for (ProductMessage pmsg : pm) {
                // Write the contents of the product queue to the buffers
                bufferWriter1.write(pmsg.toString() + (char) (10));
                bufferWriter2.write(pmsg.toString() + (char) (10));
            }
            bufferWriter1.close();
            bufferWriter2.close();
        }
    } catch (Exception ex) {
        Logger.getLogger(Utilities.class.getName()).log(Level.SEVERE, null, ex);
    }
}
```

Methods to print consumed data to a file based on regions continued.

6. MiniProject3 (driver class):

Snapshot 6-1:

```
public static void main(String[] args) {
    System.out.println("Press RETURN to exit...");
    Utilities.deleteAllOutputFiles();
    //Read product_data.txt data to list
    List<Product> productList = Utilities.readTextDataIntoArray();

    System.out.println("Simulation Starting...");
    System.out.println("Creating and Starting the ProductProducer and ProductConsumer objects")

    //get the simulation starting time.
    Date startTime = new Date();

    //Creating producer object and starting the thread.
    ProductProducer producerOne = new ProductProducer(productList);
    Thread p = new Thread(producerOne);
    p.setDaemon(true);
    p.start();

    //Creating consumer object with region N.
    ProductConsumer NConsumer = new ProductConsumer('N', producerOne, 600L);
    Thread c1 = new Thread(NConsumer);
    c1.setDaemon(true);
    c1.start();
```

Code to start simulation by reading txt file into array, by creating producer object, consumer Object, by noting the start time of the simulation.

Snapshot 6-2:

```
//Creating consumer object with region S.
ProductConsumer SConsumer = new ProductConsumer('S', producerOne, 700L);
Thread c2 = new Thread(SConsumer);
c2.setDaemon(true);
c2.start();

//Creating consumer object with region E.
ProductConsumer EConsumer = new ProductConsumer('E', producerOne, 800L);
Thread c3 = new Thread(EConsumer);
c3.setDaemon(true);
c3.start();

//Creating consumer object with region W.
ProductConsumer WConsumer = new ProductConsumer('W', producerOne, 900L);
Thread c4 = new Thread(WConsumer);
c4.setDaemon(true);
c4.start();

//Print all the data and exit when return key is pressed.
Utilities.exitAndPrintRequiredData(producerOne, NConsumer, SConsumer, EConsumer, WConsumer, startTime);
```

Code shows the creation of three more conumer object, and calls the function which exits from application when enter key is pressed, and displays all required data to console and to a file

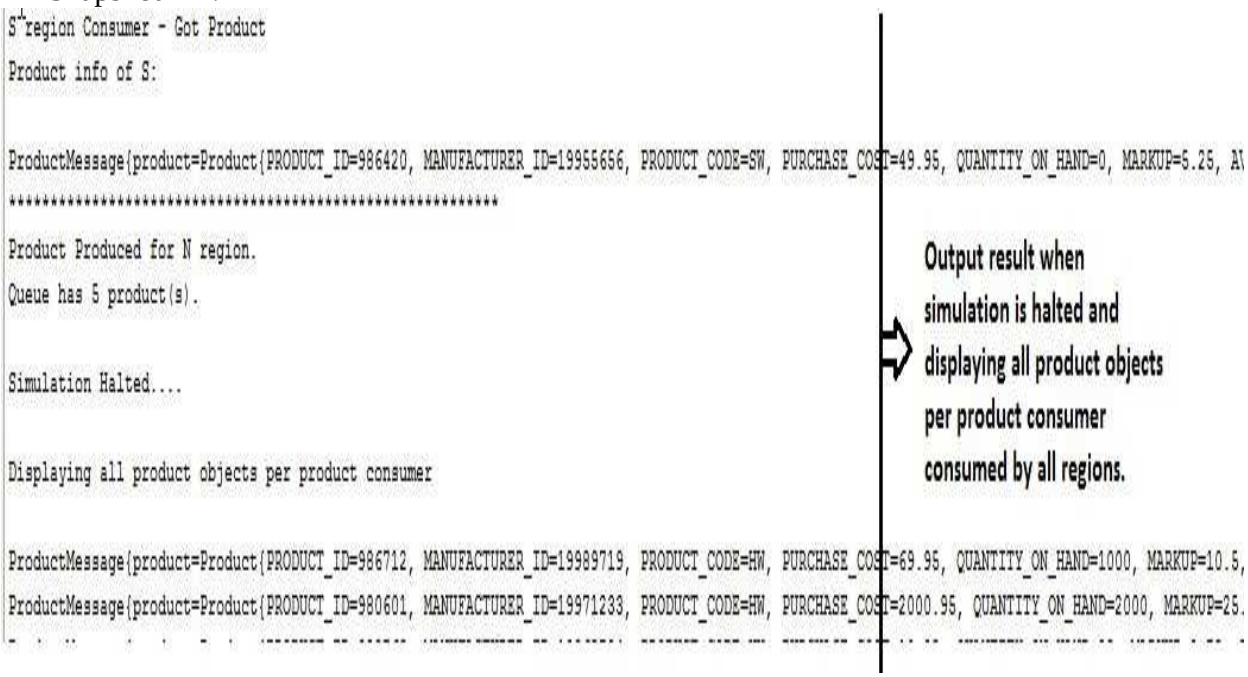
7. Output :

Snapshot 7-1:

```
run:
Press RETURN to exit...
Simulation Starting...
Creating and Starting the ProductProducer and ProductConsumer objects
Product Produced for W region.
Queue has 1 product(s).
************************************************************
W region Consumer - Got Product
Product info of W:

ProductMessage{product=Product{PRODUCT_ID=978495, MANUFACTURER_ID=19977348, PRODUCT_CODE=                QUANTITY_ON_HAND=0, MARKUP=1.0,
************************************************************
Product Produced for E region.
Queue has 1 product(s).
************************************************************
E region Consumer - Got Product
Product info of E:

ProductMessage{product=Product{PRODUCT_ID=980001, MANUFACTURER_ID=19985678, PRODUCT_CODE=SW, PURCHASE_COST=1095.0, QUANTITY_ON_HAND=800000, MARKU
************************************************************
Product Produced for E region.
```

Result of starting simulation by creating Producer and Consumer objects, displaying the queue status when produces produces the message and pushes into the queue, and the Product details after the consumer consumes themessage from queue based on region.

Snapshot 7-2:



Snapshot 7-3: