|  |  |
| --- | --- |
|  | |
| Factory Prototype | |
| **Updated** | **10.28.2018 2:46 PM** |

# **Problem 1:**

A guitar manufacturer is considering changing its paint spraying operations from a manual process, where human workers spray guitars in spraying booths, to an automated processing robot sprayer in similar booths. Each process is fed by an automated conveyor belt system.

You have been asked to design and code a multithreaded simulation to determine the amount (and efficiency) of spraying robots needed to ensure that the company is more productive and profitable than when using the manual process. You have been given the following background information:

• There are currently 20 employees in the painting department, earning an average of $40,000 per year.

• A paint worker paints on average 48 guitars in an 8 hour working day.

• A paint spraying robot costs $2,000 in running costs each week based on 40 hours of operation.

• There is an annual maintenance cost for each paint-spraying robot of $10,000.

• Guitars approach the painting line in a first in first out queue. There is space for 40 guitars to wait in the queue at a time.

• A paint worker or paint spraying robot can only spray one guitar at a time.

• If all of the manual or robot spraying booths are occupied, the guitar at the head of the queue must wait until one becomes available. They must check all the painters for availability every 6 seconds.

• When a guitar enters a spraying booth, they give up their place at the head of the queue and the next guitar moves forward to either choose a free spraying booth or wait until one becomes free. A new guitar then enters at the back of the queue.

• A guitar takes between 7 and 13 minutes to be sprayed in a manual booth.

• A guitar will take a fixed amount of time to be sprayed in a robot booth, to be tested in the simulation.

• When the factory line finishes for the day, the simulation will end when all occupied paint booths are free of guitars.

The following additional information has also been given.

• Your simulation should have four classes. These must be called HumanPainter(), RobotPainter(), Guitar(), and PaintingEfficiencyTests().

• For the purposes of the simulation, 1 minute in the real world should equal 10 milliseconds in the simulation.

• Each Guitar() object must have its own thread, representing the automated conveyor belt system behaviour.

• The HumanPainter() and RobotPainter() objects do not require their own threads but you should ensure that there is synchronisation between threads trying to access the Painter() objects.

• In consideration of performance and memory, there is no need to create a new Guitar thread for every unsprayed guitar in the factory. When a guitar object is finished being sprayed it can go to the back of the queue to represent the next guitar entering the assembly line queue.

• Consider an appropriate collection class to manage the Guitar objects, and perhaps the Painter objects.

Your PaintingEfficiencyTests() class should contain a main method that initialises an appropriate amount of HumanPainter() or RobotPainter() objects, the correct number of Guitar() objects to fill the queue, and then runs the threads in simulations representing one average day at the factory. You should run your simulation several times to discover the amount of RobotPainters(), and their painting speed, needed in order to outperform the HumanPainter() objects in terms of guitars produced vs the cost of production. At the end of the simulation you should print out the total amount of Guitars painted for each Painter() and a breakdown of the overall costs involved.

# **Approach**

The approach I took was to address this question was as a Producer Consumer problem. Since it is an example of a multi-process synchronization where Guitars (Producers) job is to generate data, put it into the buffer, and start again while the Human and Robot painters (Consumer) is taking the guitars (i.e., removing from the buffer), one unit at a time. in order to make sure that the Guitars won't add to the queue when it's full and that the Painters won't remove data from an empty buffer. To meet the requirements of the question I decided to use the BlockingQueue interface in the java.util.concurrent package which represents a queue that is thread safe to add to, and take from. I then used the output coupled with the provided background info to deduce results in terms of guitars produced vs the cost of production for both the Human and Robot painters.

# **OneBall Design**

Create an FIFO ArrayBlockingQueue of size 40

Create new Guitar Object and pass Queue

Create new Human Painter Object and pass Queue

Create new Robot Painter Object and pass Queue

Create a HashSet as Collection for Painter Objects (Unfinished)

Create a new Guitar Thread

Run the thread

Setup a counter

Loop to fill queue

Every time a slot becomes available a new guitar is added

Loop to create 20 Human Painter objects and threads

Get the Thread ID

Set the Annual Salery

Calculate Work Days per year

Calculate Daily Rate from Salery divided by Days

Calculate the Minutes in a Work day (Milliseconds = Minutes)

Setup a Random generator

Calculate random Spray Time between 7 mins and 13

Calculate the total time spent spraying by worker object

If the total time is not greater the 8 hours take next guitar

Print out the following details on each paint job

Painter ID

Current guitar ID number

The Spray time

Cumulative work minutes for this worker

Cumulative guitars painted

Output the current size of the queue

When the total time spent spraying exceeds 8 hours (Milliseconds = Minutes)

Finish worker Thread

Print - Painter ID XX painted XX guitars in XXX minutes at a cost of $XXX.XX per day

Loop to create 2 Robot Painter objects and threads

Get the Thread ID

Set the Annual Maintenance

Set the Weekly Running Costs

Calculate Work Days per year

Calculate Daily Rate from Costs divided by Days

Calculate the Minutes in a Work day (Milliseconds = Minutes)

Setup the Spray Time and adjust as required

Calculate the total time spent spraying by worker object

If the total time is not greater the 8 hours take next guitar

Print out the following details on each paint job

Machine ID

Current guitar ID number

The Spray time

Cumulative work minutes for this Machine

Cumulative guitars painted

Output the current size of the queue

When the total time spent spraying exceeds 8 hours (Milliseconds = Minutes)

Finish Machine Thread

Print - Machine ID XX painted XX guitars in XXX minutes at a cost of $XXX.XX per day

Adjust rerun and review findings to discover the amount of RobotPainters(), and their painting speed, needed in order to outperform the HumanPainter() objects in terms of guitars produced vs the cost of production.

# **Results**

* Each Human painter can paint an avg. of 48 guitars per day at an avg. cost of $154 per day
* The 20 human painters can paint an average of 960 guitars per day $3077
* Seven robot painters can produce 1127 guitars for a similar price $3069 per day
* Alternatively, six robot painters can produce a similar amount, 966 guitars for $2613
* Two robots can match the Human Painter output 962 at a rate of $877

# **Testing**

The program was run several times with different inputs to ensure the code behaves as expected for each possible execution scenario including the input of invalid and out of bound parameters.

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Test** | **Expected Result** | **Actual Result** |
| 1 | The queue mechanism works in a FIFO fashion | The queue mechanism works in a FIFO fashion | **Pass** |
| 2 | Create 20 Manual Painter objects and threads | 20 Manual Painter objects and threads are created | **Pass** |
| 3 | Create different amounts of Robot Painter objects and threads | Different amounts of Robot Painter objects and threads are created | **Pass** |
| 4 | Manual threads run and print out status | Manual threads run and print out status | **Pass** |
| 5 | Robot threads run and print out status | Robot threads run and print out status | **Pass** |
| 6 | Manually check the program outputs. | The program outputs are correct | **Pass** |

# **Source Code – PaintingEfficiency.java**

|  |
| --- |
| /\* mmcmahon\_Final\_Exam\_Q1\_PaintingEfficiency.java  \* PaintingEfficiencyTests() class should contain a main method that initialises an  \* appropriate amount of HumanPainter() or RobotPainter() objects, the correct number of  \* Guitar() objects to fill the queue, and then runs the threads in simulations representing one  \* average day at the factory.  \*/  **import** java.util.HashSet;  **import** java.util.concurrent.ArrayBlockingQueue;  **import** java.util.concurrent.BlockingQueue;  **public** **class** PaintingEfficiency {  //Create FIFO queue with size of 40 guitars  **private** **static** BlockingQueue<Integer> *queue* = **new** ArrayBlockingQueue<Integer>(  40);    **static** Guitar *Guitar* = **new** Guitar(*queue*);  **static** HumanPainter *HumanPainter* = **new** HumanPainter(*queue*);  **static** RobotPainter *RobotPainter* = **new** RobotPainter(*queue*);    **static** HashSet *painterSet* = **new** HashSet(); //Create a HashSet Collection        // PaintingEfficiency main method  **public** **static** **void** main(String[] args) {  // Create the Guitar assembly line producer thread  **new** Thread(*Guitar*).start();    // Create 20 Human Painter consumer threads  **for** (**int** i = 1; i <= 20; i++) {  **new** Thread(*HumanPainter*).start();  *painterSet*.add(Thread.*currentThread*().getId()); //Unfinished - collate results  }      **for** (**int** i = 1; i <= 2; i++) {  **new** Thread(*RobotPainter*).start();  }  }  } |

# **Source Code – Guitar.java**

|  |
| --- |
| /\* mmcmahon\_Final\_Exam\_Q1\_Guitar.java  \* Guitar object representing the automated conveyer belt system behaviour. Guitars approach  \* the painting line in a first in first out queue. There is space for 40 guitars to wait in  \* the queue at a time.  \*/  **import** java.util.Random;  **import** java.util.concurrent.ArrayBlockingQueue;  **import** java.util.concurrent.BlockingQueue;  **public** **class** Guitar **implements** Runnable{  **protected** BlockingQueue queue = **null**;  **public** Guitar(BlockingQueue queue) {  **this**.queue = queue;  }  **public** **void** run() {  **int** count = 0;  **while** ( **true** ) {  **try** {  Thread.*sleep*(0); // no sleep  queue.put(count);  count++;  } // end try  // if sleeping thread interrupted, print stack trace  **catch** (InterruptedException exception) {  exception.printStackTrace();  } // end catch  } // end for  }  } |

# **Source Code – HumanPainter.java**

|  |
| --- |
| /\* mmcmahon\_Final\_Exam\_Q1\_HumanPainter.java  \* Represents an employees in the painting department painting on average 48 guitars in an 8 hour working day.  \* A paint worker or paint spraying robot can only spray one guitar at a time. A guitar takes between 7 and 13 minutes  \* to be sprayed in a manual booth.  \*/  **import** java.util.Random;  **import** java.util.concurrent.ArrayBlockingQueue;  **import** java.util.concurrent.BlockingQueue;  **import** java.util.concurrent.LinkedBlockingQueue;  **public** **class** HumanPainter **implements** Runnable{  **protected** BlockingQueue queue = **null**;  **public** HumanPainter(BlockingQueue queue) {  **this**.queue = queue;  }  **public** **void** run() {  **long** painterT = Thread.*currentThread*().getId();  **int** count = 0;  **int** sprayTime = 0;  **int** totalTime = 0;  **double** Salery = 40000;  **double** WorkDays = (52\*5);  **double** DailyRate = (Salery/WorkDays);  **int** WorkMinutes = (8 \* 60) \* 10;    **do** {  **try** {  Random generator = **new** Random(); // Create a random generator  **int** max = 130; // Max 13 minutes for manual spray  **int** min = 70; // Min 7 minutes for manual spray  sprayTime = (generator.nextInt((max - min) + 1) + min);  Thread.*sleep*(sprayTime); //Random Sleep  totalTime += sprayTime;  Integer value = (Integer) queue.take(); // Take next guitar in queue  System.***out***.println("Painter ID " + painterT  + " is spraying guitar number: " + value  + "\nSpray time was " + sprayTime  + "\nCumulative work minutes: " + totalTime  + "\nCumulative guitars painted: " + count  + "\nCurrent queue size is: "+ queue.size() + "\n");  count++;  } // end try  // if sleeping thread interrupted, print stack trace  **catch** (InterruptedException exception) {  exception.printStackTrace();  } // end catch  } **while** (totalTime <= WorkMinutes);  System.***out***.println("Painter ID " + painterT + " painted " + count + " guitars in " + (totalTime/10) + " minutes at a cost of $" + DailyRate +" per day\n");  } // end method run } |

# **Source Code – RobotPainter.java**

|  |
| --- |
| /\*mmcmahon\_Final\_Exam\_Q1\_RobotPainter.java  \* Represents an spraying robot in the painting department. A paint spraying robot costs $2,000 in running costs  \* each week based on 40 hours of operation. There is an annual maintenance cost for each paint-spraying robot of  \* $10,000. A paint spraying robot can only spray one guitar at a time. A guitar will take a fixed amount of time  \* to be sprayed in a robot booth, to be tested in this simulation.  \*/  **import** java.util.Random;  **import** java.util.concurrent.ArrayBlockingQueue;  **import** java.util.concurrent.BlockingQueue;  **import** java.util.concurrent.Executors;  **import** java.util.concurrent.locks.Lock;  **import** java.util.concurrent.locks.ReentrantLock;  **public** **class** RobotPainter **implements** Runnable{  **protected** BlockingQueue queue = **null**;  **public** RobotPainter(BlockingQueue queue) {  **this**.queue = queue;  }  **public** **void** run() {  **long** machineT = Thread.*currentThread*().getId();  **int** count = 0;  **int** sprayTime = 0;  **int** totalTime = 0;  **double** annualMaintaince = 10000;  **double** weeklyRunCosts = 2000;  **double** WorkDays = (52\*5);  **double** DailyRate = ((annualMaintaince/WorkDays) + (weeklyRunCosts/5));  **int** WorkMinutes = (8 \* 60) \* 10;    **do** {  **try** {  sprayTime = (10);  Thread.*sleep*(sprayTime); //Random Sleep  totalTime += sprayTime;  Integer value = (Integer) queue.take(); // Take next guitar in queue  System.***out***.println("Machine ID " + machineT  + " is spraying guitar number: " + value  + "\nSpray time was " + sprayTime  + "\nCumulative work minutes: " + totalTime  + "\nCumulative guitars painted: " + count  + "\nCurrent queue size is: "+ queue.size() + "\n");  count++;  } // end try  // if sleeping thread interrupted, print stack trace  **catch** (InterruptedException exception) {  exception.printStackTrace();  } // end catch  } **while** (totalTime <= WorkMinutes);  System.***out***.println("Machine ID " + machineT + " painted " + count + " guitars in " + (totalTime/10) + " minutes at a cost of $" + DailyRate +" per day\n");  } // end method run } |

# **Problem 2:**

The same guitar manufacturer has asked you to assist with their warehouse management system. Guitar assembly workers select parts from the online catalogue and an assistant picks them from the shelf for the assembly worker to build guitars with. Records of part levels are kept in a text file. Your prototype will contain a small text file for testing purposes. It should contain the following information (without the headings):

Part value amount

Body 200 100

Neck 100 150

Bridge 50 80

Tuners 40 70

Pickups 75 25

* The part file must be opened by a class named Parts, which reads the values into an appropriate array or list.
* Each interaction is handled by a separate thread. The thread will check the levels of the requested part against the amount required by the assembly worker. If there is enough of the part, the amount should be decreased and the array updated. If there is not enough of the part, the user must be given an appropriate error message.
* Only one thread can access the Parts array at a time to ensure levels are accurate. E.g. if one worker requests 20 bodies and another worker requests 10 at the same time from a different interaction, the first worker should get the 20 bodies and the second worker should be given a message telling them that there is not enough parts.
* The part amounts should never reach a minus figure. This will require synchronization between threads.
* Write a class called Transaction that accepts a part description and a quantity in the constructor. The Transaction class should check part amounts and update the array once its thread is executed. Each transaction should take 2ms to complete. The current part amounts should be displayed in the console after each transaction.
* Write a test class called TransactionTest that initialises a Parts object, creates 10 transactions and executes them at the same time. Be sure to include a scenario where two Transactions compete for a particular part.
* Write a second test class that generates and executes transactions with random item description and random quantity (between 1 and 20) until all parts amounts have reached zero.

# **Approach**

NOTE: I was unable to complete this question as I ran out of time before the deadline.

# **Warehouse Management System Design**

Initialise a Part Array

Initialise part, value, amount

Create a Parts Constructor

Set part

Get part

Set value

Get value

Set amount

Get amount

Read record.txt to Array

Get the PartsArray

Print the Array

Return a parts string

Transaction constructor accepts a part description and a quantity

Check the amount of parts

Update the part amount

Display current part amounts display

TransactionTest that initialises a Parts object

Create and execute 10 transactions

Create test class that generates and executes randon part, amount transactions

# **Testing**

The program was run several times with different inputs to ensure the code behaves as expected for each possible execution scenario including the input of invalid and out of bound parameters.

|  |  |  |  |
| --- | --- | --- | --- |
| **#** | **Test** | **Expected Result** | **Actual Result** |
| 1 |  |  |  |
| 2 |  |  |  |
| 3 |  |  |  |
| 4 |  |  |  |
| 5 |  |  |  |
| 6 |  |  |  |
| 7 |  |  |  |

# **Source Code – Parts.java**

|  |
| --- |
| /\* mmcmahon\_Final\_Exam\_Q1\_Parts.java  \* The part file is opened by a class named Parts, which reads the values into an appropriate array or list.  \*  \*/  **import** java.io.BufferedReader;  **import** java.io.FileNotFoundException;  **import** java.io.FileReader;  **import** java.io.IOException;  **import** java.util.Arrays;  **import** java.util.StringTokenizer;  **public** **class** Parts {    Parts[] partsArray = **new** Parts[5]; // Initialise a Part Array  **private** String part; // name of the part  **private** String line; // line for file to array  **private** **int** value; // value of this part  **private** **int** amount; // amount available of this part  **int** count = 0;  // Part constructor  **public** Parts(String partName, **int** valueAmt, **int** amountTot) {  **this**.part = partName;  **this**.value = valueAmt;  **this**.amount = amountTot;  }  // ...set part  **public** **void** setPart(String partName) {  part = partName;  }  // ...get part  **public** String getPart() {  **return** part;  }  // ...set value  **public** **void** setValue(**int** valueAmt) {  value = valueAmt;  }  // ...get value  **public** **int** getValue() {  **return** value;  }  // ...set amount  **public** **void** setAmount(**int** amountTot) {  amount = amountTot;  }  // ...get amount  **public** **int** getAmount() {  **return** amount;  }    //read record.txt to Array  **public** **void** readFileToArray()  {  StringTokenizer tokenizer;  **try**  {  // Open the input file  BufferedReader inFile = **new** BufferedReader(**new** FileReader("c:/temp/record.txt"));  line = inFile.readLine();  **while** (line != **null**)  {  //Split text file by lines  tokenizer = **new** StringTokenizer (line);  **try**  {  part = tokenizer.nextToken(); // First element is part name  amount = Integer.*parseInt* (tokenizer.nextToken()); // second element is amount  value = Integer.*parseInt* (tokenizer.nextToken()); // third element is value  partsArray[count++] = **new** Parts (part, amount, value); //Create an Array of Objects  }  **catch** (NumberFormatException exception)  {  System.***out***.println ("Error in input. Line ignored:");  System.***out***.println (line);  }  line = inFile.readLine();  }  inFile.close();  }  **catch** (FileNotFoundException exception)  {  System.***out***.println ("The file was not found.");  }  **catch** (IOException exception)  {  System.***out***.println (exception);  }  }    **public** Parts[] getPartsArray(){  **return** partsArray;  }    **public** **void** printArray() {  **for** (**int** i = 0; i < partsArray.length; i++)  System.***out***.println(partsArray[i]);  }  // Return a parts string  **public** String toString() {  **return** part + " " + value + " " + amount + "";  }  **public** **void** modifyArrayTest(String partName, **int** valueAmt, **int** amountTot) {  **boolean** found = **false**;  **for** (**int** j = 0; j < partsArray.length; j++) {  **if** ( partsArray[j].part == partName ) {  found = **true**;  }  }  **if** (found) {  System.***out***.println( "\nThe array contains the string: " + partName );  } **else** {  System.***out***.println( "\nThe array does not contains the string: " + partName );    }    }  } |

# **Source Code – Transaction.java**

|  |
| --- |
| **import** java.util.Arrays;  **public** **class** Transaction {  **private** String part; // part of the part  **private** **int** amount; // amount available of this part      //Constructor accepts a part description and a quantity  **public** Transaction(String partName, **int** amountTot) {  part = partName;  amount = amountTot;  }    //Check the amount of parts  **public** **void** checkPartAmount(){    }    //Update the part amount  **public** **void** updatePartAmount(){    }    //Display current part amounts display  **public** **void** dispCurrentPartAmt(){  }  } |

# **Source Code – TransactionTest.java**

|  |
| --- |
| /\*initialises a Parts object, creates 10 transactions and executes them at the same time.  \* Be sure to include a scenario where two Transactions compete for a particular part\*/  **public** **class** TransactionTest {    **public** **static** **void** main(String[] args) {  Parts p1 = **new** Parts(**null**, 0, 0);  p1.readFileToArray();  p1.printArray();  p1.modifyArrayTest("Body", 0, 101);  }  } |

# **References:**

*Advanced Java: Multi-threading Part 7 - Producer-Consumer*. (2016). *YouTube*. Retrieved 16 October 2016, from <https://www.youtube.com/watch?v=Vrt5LqpH2D0>

(2016). Retrieved 16 October 2016, from [http://javarevisited.blogspot.ie/2012/02/producer-consumer-design- pattern-with.html](http://javarevisited.blogspot.ie/2012/02/producer-consumer-design-%20pattern-with.html)

*Blockingqueue*. (2016). *Tutorials.jenkov.com*. Retrieved 16 October 2016, from <http://tutorials.jenkov.com/java-util-concurrent/blockingqueue.html>

Deitel,. (2016). *Cs.fit.edu*. Retrieved 16 October 2016, from <https://cs.fit.edu/~mmahoney/cis5100/examples/ch23/fig23_15_16/Buffer.java>

Deitel,. (2016). *BlockingBuffer*. *Cs.fit.edu*. Retrieved 16 October 2016, from <https://cs.fit.edu/~mmahoney/cis5100/examples/ch23/fig23_15_16/BlockingBuffer.java>

Gupta, L., Wates, R., Hoover, M., & Hoover, M. (2012). *Java ThreadPoolExecutor and BlockingQueue Example - HowToDoInJava*. *HowToDoInJava*. Retrieved 16 October 2016, from <http://howtodoinjava.com/core-java/multi-threading/how-to-use-blockingqueue-and-threadpoolexecutor-in-java/>

*How to use Java BlockingQueue with multiple threads*. (2016). *Java2s.com*. Retrieved 16 October 2016, from <http://www.java2s.com/Tutorials/Java/Thread/How_to_use_Java_BlockingQueue_with_multiple_threads.htm>

Line, J. (2013). *Java Practice Assignment #3 - Assembly Line - How to Program with Java*. *How to Program with Java*. Retrieved 16 October 2016, from <https://howtoprogramwithjava.com/java-practice-assignment-3/>

pool?, H. (2016). *How to get thread id from a thread pool?*. *Stackoverflow.com*. Retrieved 16 October 2016, from <http://stackoverflow.com/questions/3294293/how-to-get-thread-id-from-a-thread-pool/18337222>

*Producer Consumer Design Pattern*. (2016). Retrieved 16 October 2016, from <http://ttp://www.cs.colorado.edu/~kena/classes/5828/s12/assignments/homework-2.html>

Sarma, S. & profile, V. (2012). *Solving a Producer-Consumer Problem in Java*. *Javawithswaranga.blogspot.ie*. Retrieved 16 October 2016, from <http://javawithswaranga.blogspot.ie/2012/05/solving-producer-consumer-problem-in.html>