**Studying the Non-Deterministic Behavior**

**of a Producer-Consumer System**

COMP4721 Assignment #1

Michael Anderson

201449410

Sept 28, 2017

COMP4721

Ashoke Deb

**Table of Contents:**

**User’s Guide**

* 1. **Overview……………………………………………………………………………………………………………………………………3**
  2. **System Requirements………………………………………………………………………………………………………3**
  3. **Operating Guide…………………………………………………………………………………………………………………4**
  4. **Instruction (What to type)………………………………………………………………………………5-6**

**Logical Analysis**

* 1. **Logical Structure of the System………………………………………………………………………6**
  2. **Black Box Definition……………………………………………………………………………………………………7**

**Dictionary**

**2.1 Dictionary of Functions………………………………………………………………………………………8-9**

**2.1.1 Built-in Functions………………………………………………………………………………………8**

**2.1.2 User-defined Functions……………………………………………………………………………9**

**2.2 Dictionary of Variables…………………………………………………………………………………………10**

**Java Code & Testing**

**3.1 The Code…………………………………………………………………………………………………………………………11-14**

**3.2 Run-time output………………………………………………………………………………………………………………15**

**3.3 Producer() Function Test………………………………………………………………………………………16**

**3.4 Consumer() Function Test………………………………………………………………………………………17**

**3.5 Stop() Function Test…………………………………………………………………………………………………17**

**3.6 UserProducerDecider() Function Test…………………………………………………………18**

**3.7 Scheduler() Function Test……………………………………………………………………………………19**

**3.8 toString() Function Test………………………………………………………………………………20-21**

**Issues and Limitations**

**4.1 Error Handling…………………………………………………………………………………………………………………22**

**4.2 Generality……………………………………………………………………………………………………………………………22**

**User’s Guide**

**0.1 Overview: Non-determinisitc behavior of Producer-Consumer System**

The goal of the study was to analyze and create an imitation of a non-deterministic system. A given system, m, is non-deterministic when the output of m is not directly tied to the input of m. In other words, the handling of the input is inherently random.

This was achieved by using Java’s Random.util and random number generation tools.

Given the user’s input, the system calculates a random value between 1 – userInput, using a Seed value that is derived from the milliseconds on the actively running system’s clock.

If the random value is even, a Producer() function is called, which generates a new value for the global variable sequenceValue (in a way described by the user).

If the random value is odd, a Consumer() function is called, which stores the value of the global variable and prints it to the user.

When the user chooses to end the program, a summary of the results is printed.

**0.2 System Requirements**

The program may be access and modified using Java 7 and newer, through any function IDE. The program can be ran through the system console (CMD.exe) or through an IDE with runtime support.Java 7 may be ran on Microsoft Windows XP and newer operating systems, as well as an Intel-based Mac running Mac OS X 10.7.3 or later operating system. For full Java 7 system requirements visit <http://www.oracle.com/technetwork/java/javase/config-417990.html>

**0.3 Operating Guide**

The file name of the program is ProducerConsumer.java and can be saved under any supporting public directory. For the sake of testing, it was saved in the directory of:

C:\Users\Michael\Documents\4721 A1\src\com\company\ProducerConsumer.java

To locate and access the program, the user can open the Start menu and navigate to the Search feature. Typing “cmd” and pressing enter will open the system’s Command Line. The user may run the program by navigating to the directory in which they have the program saved, and then then executing it. For the example above, the user would enter

“cd Documents … cd 4721 A1 … cd src\com\company”

To compile and execute the program the user types:

Compile:

“javac ProducerConsumer.java”

Execute:

“java –classpath com.company.ProducerConsumer”

**0.4 Instructions (What to type)**

When first running the program, it will prompt you with the following statement:

*Please enter the Range of the values generated for your sequence*

What you are required to enter here is a natural number that you wish to be the Range of the values generated randomly by the program. You can type any natural number.

You will then be prompted with:

*Please select one of the following options, by entering the corresponding integer value:*

1. *– increment the value of the global variable by 1 for each Producer call*
2. *– randomly generate the value of the global variable for each Producer call*

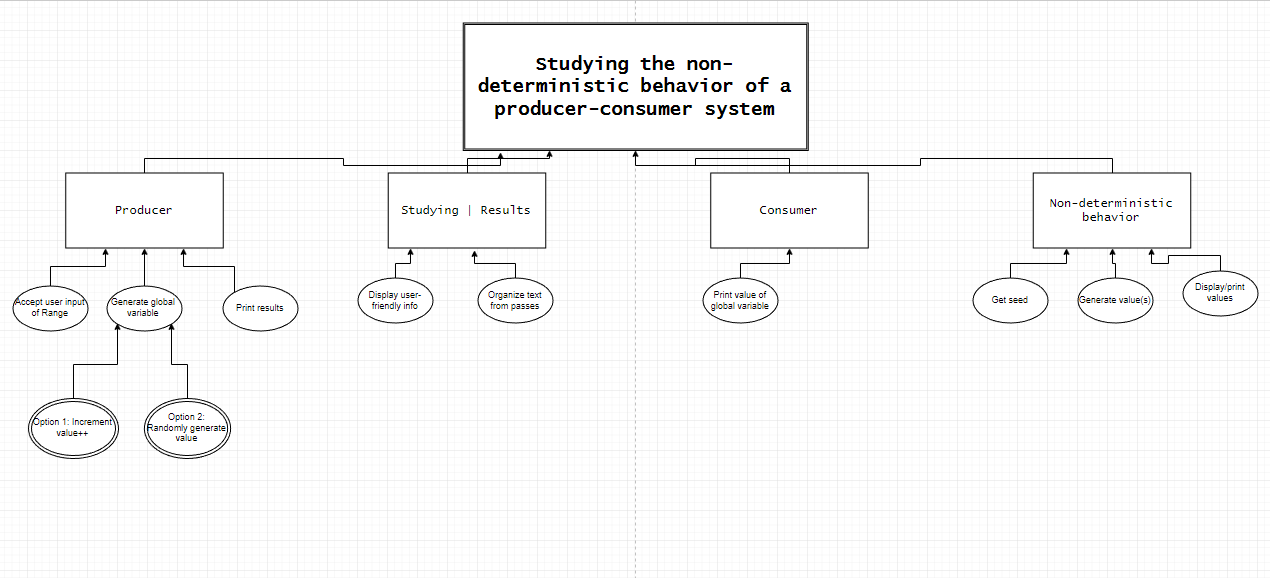
You can enter either “1” or “2”. Nothing more, nothing less.

The program will then continuously run on a loop until you enter *anything at all* to prompt a stop.

**Logical Analysis**

**1.1 Logical Structure of the System**

Below is the top-down logic follow when developing the logical structure of the code.

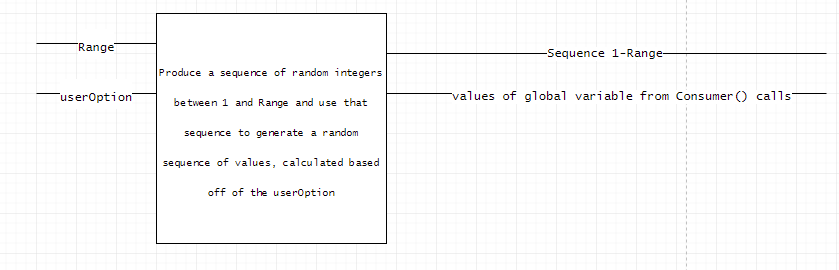


**1.2 Black Box Definition**

**Input:** As shown in the diagram, the input consists of a user selected Range and userOption. The range will dictate the

maximum values of the sequence produced to decide the Producer() and Consumer() calls, and the userOption will dictate how the Producer function generates the value of the global variable.

**Output:** The output will consist of the sequence the was randomly generated (with maximum values of *Range)*, as well as the values of the global variable that were printed by Consumer() function calls.



**Dictionary**

**2.1 Dictionary of Functions**

Within the program, many functions are used. Some are defined within the program while others are built-in Java functions:

**2.1.1 Built-in Functions:**

**println()** – Prints the strings contained in the argument to the user.

**newSingleThreadScheduledExecutor()** – creates a new executor that will, on a loop, run the contents of the executor.

**hasNext()** – returns whether or not the prefix has a pending input.

**nextInt()** – returns the next integer in the prefix.

**add()** – adds the argument to the queue described in the prefix.

**size()** – returns the size of the queue in the prefix.

**remove()** – removes the first-entered element in the queue in the prefix.

**LinkedList<Integer>()** – creates a new LinkedList of integers.

**Scanner()** – creates a new Scanner objects that reads from the argument.

**Random()** – creates a Random Number Generator that uses the argument as its seed value.

**currentTimeMillis()** – returns the current value of the milliseconds on the system clock.

**2.1.2 User-defined Functions:**

**Producer()** – generates the value of the global variable sequenceVariable based off of the userOption.

**Consumer()** – calls the toString function with an argument of the current global variable.

**UserProducerDecider()** – uses the value of userOption to determine the method in which the program will generate the value of the global variable sequenceValue.

**Stop()** – when the user prompts the program to stop, calls the toString() and exits the program.

**toString()** – stores the random number generator values in a queue. Stores the values of the sequenceValue global variable at the time of a Consumer() function call in a queue. Prints the contents of both queues when called by the Stop() function.

**Scheduler()** – generates a random value which determines whether the Scheduler calls the Producer() or Consumer() functions

**2.2 Dictionary of Variables**

The following is a list of all variables used within the program code:

**sequenceValue –** Global integer value, initially set to 1, and then generated depending on userOption.

**RNGValues –** Global queue containing the values randomly generated by the random number generator.

**sequenceValueArray –** Global queue containing the values of the sequenceValue global variable at the time of Consumer() calls.

**Range –** User inputted value limiting the maximum value of the numbers generated by the random number generator.

**userOption –** User inputted value determining the method in which the Producer generates subsequent values of sequenceValue.

**executorService –** Instance of the ScheduledExecutorService object used to loop the call of the Scheduler() function.

**in –** Instance of the Scanner object.

**RNG –** Instance of the Random object.

**seed –** Value generated based off of the current milliseconds on the system clock used to alter the values produced by the random number generator.

**System –** variable indicating the current running system.

**sender –** Integer value used to differentiate the function that called the toString() function.

**prodConDecider –** Integer value used to determine whether the Producer or Consumer functions are called.

**seqArraySize –** Integer value containing the current size of the sequenceValueArray queue.

**rngArraySize –** Integer value containing the current size of the RNGValue queue.

**i –** Integer value used to iterate through a loop.

**val –** Integer value used to contain a value from either the sequenecValueArray or RNGValues queues that is next to be printed.

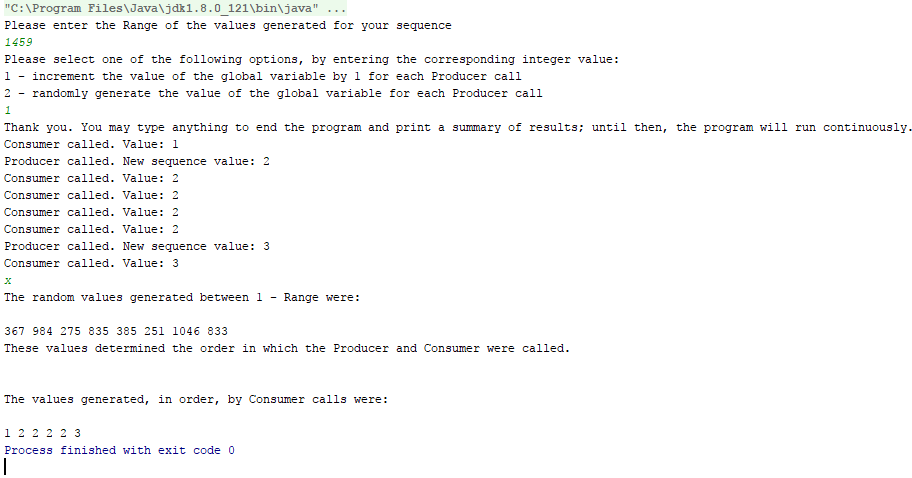
**Java Code & Testing**

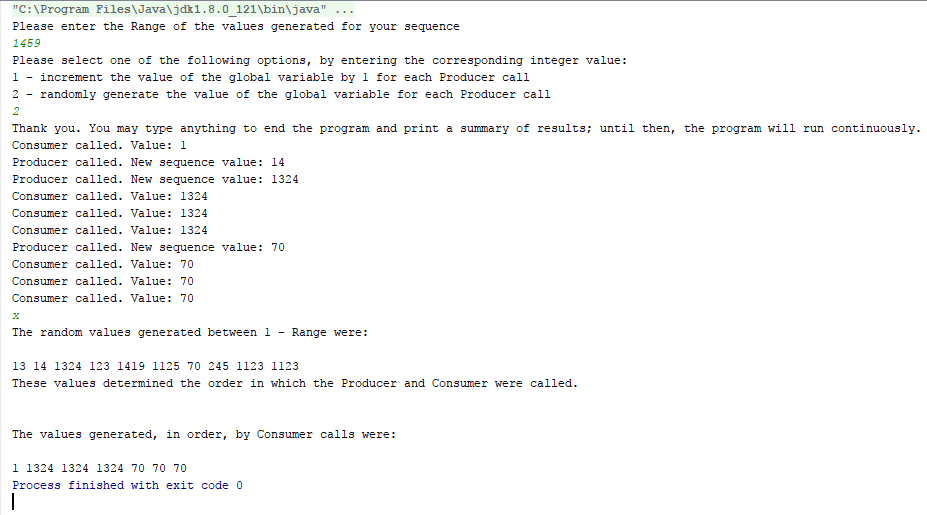
**3.1 The Code**

Below is the full Java code for the program:

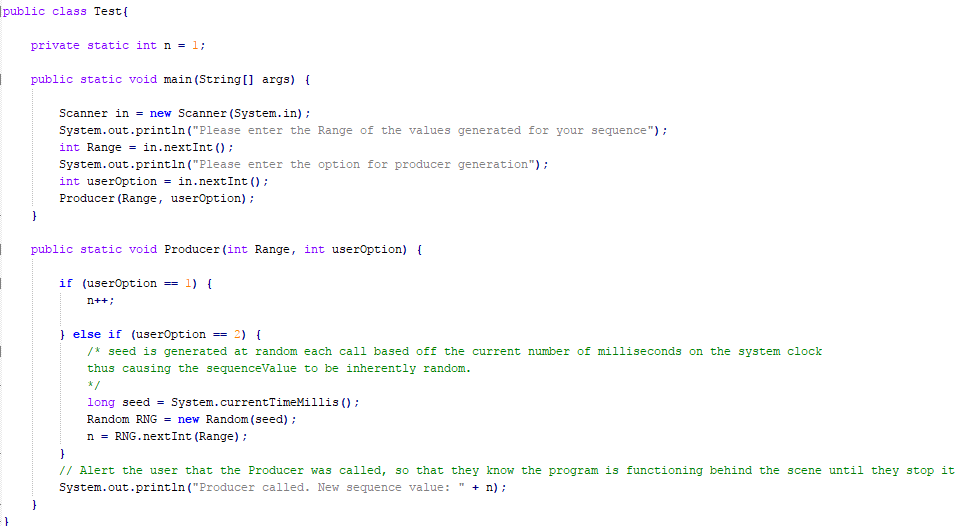
**package** com.company;  
**import** java.util.LinkedList;  
**import** java.util.Queue;  
**import** java.util.Random;  
**import** java.util.Scanner;  
**import** java.util.concurrent.Executors;  
**import** java.util.concurrent.ScheduledExecutorService;  
**import** java.util.concurrent.TimeUnit;  
  
  
**public class** ProducerConsumer {  
 */\*  
 global variables to be altered through the random calls of Producer and Consumer.  
 \*/* **private static int** *sequenceValue* = 1;  
 **private static** Queue<Integer> *RNGValues* = **new** LinkedList<Integer>();  
 **private static** Queue<Integer> *sequenceValueArray* = **new** LinkedList<Integer>();  
  
 */\*  
 main method. gathers initial user input of desired Range, and prompts them to choose which Producer option they prefer  
 \*/* **public static void** main(String[] args) {  
 Scanner in = **new** Scanner(System.***in***);  
 System.***out***.println(**"Please enter the Range of the values generated for your sequence"**);  
 **int** Range = in.nextInt();  
 **int** userOption = *UserProducerDecider*();  
 *// ScheduledExecutorService allows the program to, on an interval, loop the call of a single function.* **final** ScheduledExecutorService executorService = Executors.*newSingleThreadScheduledExecutor*();  
  
 *// periodically (every 2 seconds) re-run the Scheduler() function until user inputs a value to stop the program.* **do** {  
 executorService.scheduleAtFixedRate(**new** Runnable() {  
 @Override  
 **public void** run() {  
 *Scheduler*(Range, userOption);  
 }  
 }, 0, 2, TimeUnit.***SECONDS***);  
 } **while**(!in.hasNext());  
 **if**(in.hasNext()){  
 *Stop*();  
 }  
  
 }  
  
 *// Function to gather the user's desired method of generating the global variable sequenceValue* **private static int** UserProducerDecider() {  
 Scanner in = **new** Scanner(System.***in***);  
 System.***out***.println(**"Please select one of the following options, by entering the corresponding integer value:"**);  
 System.***out***.println(**"1 - increment the value of the global variable by 1 for each Producer call"**);  
 System.***out***.println(**"2 - randomly generate the value of the global variable for each Producer call"**);  
 **int** userOption = in.nextInt();  
 *// ensure that userOption is a valid selection* **while**(userOption > 2) {  
 System.***out***.println(**"Invalid selection. Please choose again."**);  
 userOption = in.nextInt();  
 }  
 *// accept user input and move forward* System.***out***.println(**"Thank you. You may type anything to end the program and print a summary of results; until then, the program will run continuously."**);  
  
 *// return the user decision to userOption variable to begin Scheduler loop.* **return** userOption;  
 }  
  
 */\* Producer method will, depending on userOption, generate a new value of the global variable sequenceValue  
 each time the function is called. Currently coded is the ability to increment the value by 1 each call,  
 or to have the value be entirely random (still limited by the user inputted Range from before, to avoid  
 overflow values).  
 To add subsequent methods of generation, simply write another if-else statement with a userOption  
 condition of 3 or greater, and have that value be passed into Producer() by adding a new prompt option in UserProducerDecider()  
 \*/* **private static void** Producer(**int** Range, **int** userOption) {  
  
 **if** (userOption == 1) {  
 *sequenceValue*++;  
  
 } **else if** (userOption == 2) {  
 */\* seed is generated at random each call based off the current number of milliseconds on the system clock  
 thus causing the sequenceValue to be inherently random.  
 \*/* **long** seed = System.*currentTimeMillis*();  
 Random RNG = **new** Random(seed);  
 *sequenceValue* = RNG.nextInt(Range);  
 }  
 *// Alert the user that the Producer was called, so that they know the program is functioning behind the scene until they stop it* System.***out***.println(**"Producer called. New sequence value: "** + *sequenceValue*);  
 }  
  
 */\*  
 The consumer function is trivial. As the sequenceValue is stored globally, the Consumer doesn't accept any args.  
 It simply calls the toString method with the 'sender' value of 2, alerting the toString method of what to  
 do with the incoming sequenceValue.  
 \*/* **private static void** Consumer() {  
 *toString*(2, *sequenceValue*);  
 }  
  
 */\* The Scheduler() accepts both the userOption and Range, containing user inputted integers.  
 This function is called on a two-second time interval, and controls the flow of the entire program.  
 A random value is calculated, using a randomly generated seed, and based upon that value, the Producer or  
 Consumer functions are called.  
 The randomly generated numbers are also stored in an array, by calling the toString function below.  
 \*/* **private static void** Scheduler(**int** Range, **int** userOption) {  
 */\* seed is generated at random each call based off the current number of milliseconds on the system clock  
 thus causing the sequenceValue to be inherently random.  
 \*/* **long** seed = System.*currentTimeMillis*();  
 Random RNG = **new** Random(seed);  
 **int** prodConDecider = RNG.nextInt(Range);  
  
 **if** (prodConDecider % 2 == 0) {  
 *Producer*(Range, userOption);  
 } **else** *Consumer*();  
  
 *toString*(1, prodConDecider);  
 }  
  
 */\*  
 The toString function is called from several different areas in the program, and based on where it  
 is called from, it must print different fragments of data and store its arguments in different places,  
 either global or local. Because of this, a sender argument is used, to determine what the toString method will do.  
 It is contained in a single function because each and every sender value still causes it to output data to the User,  
 just in different formats.  
 \*/* **private static void** toString(**int** sender, **int** sequenceValue) {  
 *// sender = 1 indicates the data is coming from the Scheduler. Thus, arg[1] is the RNG value to determine Producer/Consumer calls* **if** (sender == 1) {  
 *RNGValues*.add(sequenceValue);  
 }  
 */\* sender = 2 indicates data is coming from the Consumer.  
 The value of arg[1] is then the value of the sequenceValue global variable at the time of  
 the Consumer() call, and is printed to the user immediately. It also, however, is stored in  
 a sequenceValueArray (a queue), to be printed in order upon ending the program to the user.  
 \*/* **else if** (sender == 2) {  
 *sequenceValueArray*.add(sequenceValue);  
 System.***out***.println(**"Consumer called. Value: "** + sequenceValue);  
 }  
 */\* sender = 3 indicates the data is coming from the Stop() function.  
 At this time, the user has demanded that the program terminate, and the results be printed in summary.  
 The values of both the RNGValue and sequenceValue queues are printed in user-friendly format.  
 \*/* **else if** (sender == 3) {  
 **int** seqArraySize = *sequenceValueArray*.size();  
 **int** rngArraySize = *RNGValues*.size();  
 System.***out***.println(**"The random values generated between 1 - Range were: \n"**);  
 **for**(**int** i=0; i<rngArraySize; i++) {  
 **int** val = *RNGValues*.remove();  
 System.***out***.print(val + **" "**);  
 }  
 System.***out***.println(**"\nThese values determined the order in which the Producer and Consumer were called. \n\n"**);  
 System.***out***.println(**"The values generated, in order, by Consumer calls were: \n"**);  
 **for**(**int** i=0; i<seqArraySize; i++) {  
 **int** val = *sequenceValueArray*.remove();  
 System.***out***.print(val + **" "**);  
 }  
 }  
 }  
 *// When this function is called, prompt the toString function to print results, then exit the program.* **private static void** Stop() {  
 *toString*(3, 0);  
 System.*exit*(0);  
 }  
}

**3.2 Run-time Output**

ProducerConsumer.java ran with Option 1 (increment sequenceValue by 1) chosen.

ProducerConsumer.java ran with Option 2 (randomly generate the value of sequenceValue) chosen.

**3.3 Producer() Function Test**

**Using the following segment of code, the Producer()function was tested.**

**Output:**

C:\Users\Michael\Desktop>java Test

Please enter the Range of the values generated for your sequence

1000

Please enter the option for producer generation

1

Producer called. New sequence value: 2

C:\Users\Michael\Desktop>java Test

Please enter the Range of the values generated for your sequence

9999

Please enter the option for producer generation

2

Producer called. New sequence value: 568

**3.4 Consumer() Function Test**

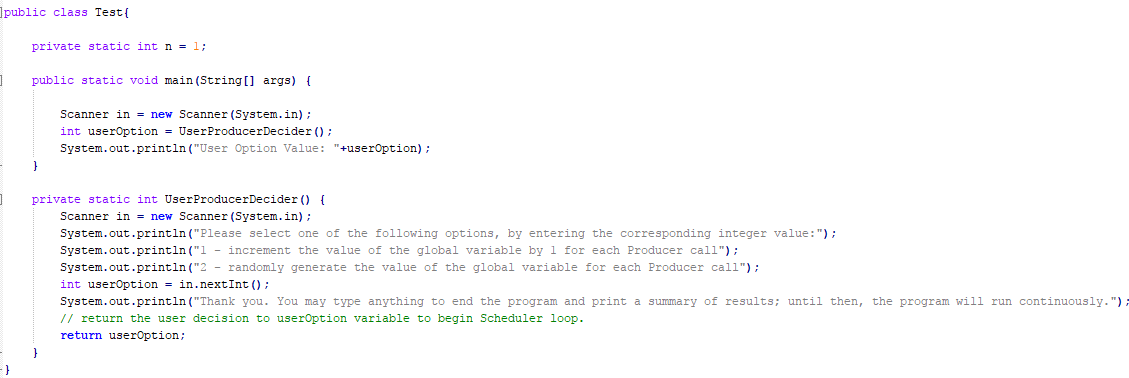
The Consumer() function is trivial and is reliant on the toString() function, tested below.

**3.5 Stop() Function Test**

The Stop() function is trivial and is reliant on the toString() function, tested below.

**3.6 UserProducerDecider() Function Test**

**Using the following segment of code, the UserProducerDecider()functionwas tested.**



**Output:**

Please select one of the following options, by entering the corresponding integer value:

1 - increment the value of the global variable by 1 for each Producer call

2 - randomly generate the value of the global variable for each Producer call

1

Thank you. You may type anything to end the program and print a summary of results; until then, the program will run continuously.

User Option Value: 1

C:\Users\Michael\Desktop>java Test

Please select one of the following options, by entering the corresponding integer value:

1 - increment the value of the global variable by 1 for each Producer call

2 - randomly generate the value of the global variable for each Producer call

2

Thank you. You may type anything to end the program and print a summary of results; until then, the program will run continuously.

User Option Value: 2

**3.7 Scheduler()Function Test**

**Using the following segment of code, the Scheduler()function was tested.**

**Output:**

C:\Users\Michael\Desktop>java Test

Seed: 1506473645158

prodConDecider: 1988

Producer called

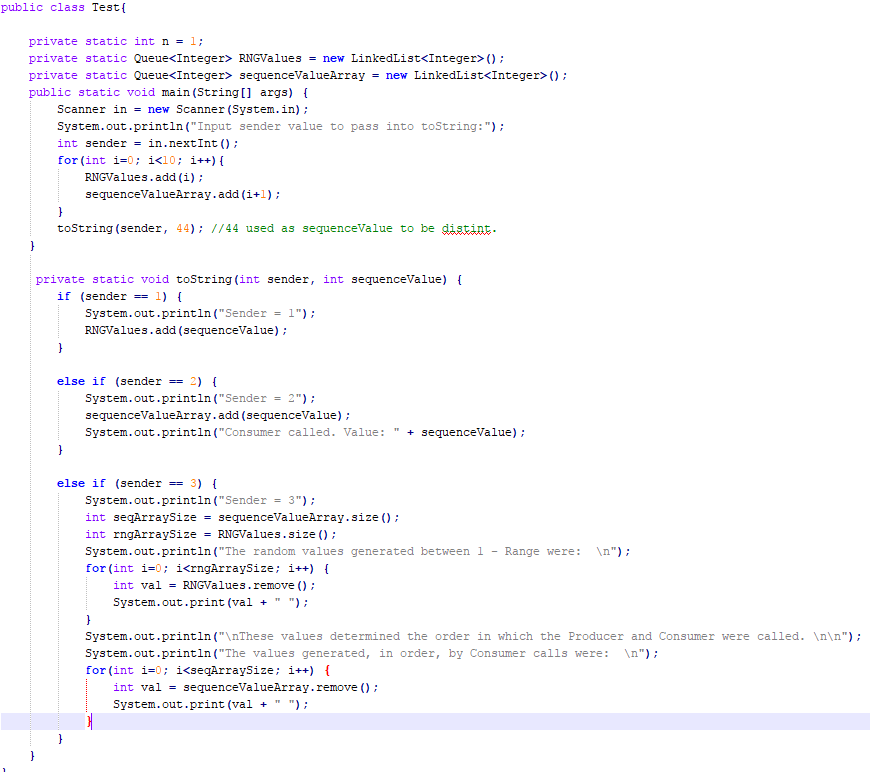
C:\Users\Michael\Desktop>java Test

Seed: 1506473657263

prodConDecider: 2463

Consumer called.

**3.8 toString() Function Test**

**Using the following segment of code, the toString()function was tested.**

**Output:**

Input sender value to pass into toString:

1

Sender = 1

C:\Users\Michael\Desktop>java Test

Input sender value to pass into toString:

2

Sender = 2

Consumer called. Value: 10

C:\Users\Michael\Desktop>java Test

Input sender value to pass into toString:

3

Sender = 3

The random values generated between 1 - Range were:

0 1 2 3 4 5 6 7 8 9

These values determined the order in which the Producer and Consumer were called.

The values generated, in order, by Consumer calls were:

1 2 3 4 5 6 7 8 9 10

**Issues and Limitations**

**4.1 Error Handling**

Given time restraints, certain areas of error handling could be more prevalent. There is the possibility that the user inputs an integer value too large which causes a system overflow, however unlikely.

**4.2 Generality**

As with any program, it can be more general to allow for easier adaptation. Adaptation was taken into account, as seen with the UserProducerDecider() function and the Producer() function, which allow for easy additions or changes to be made to alter the way in which the program updates the value of the global variable. If the intended use was to alter the variable based off of hardware input or input from a file, it could be done there without having to alter additional parts of the program. The time where this is less prevalent is the way in which the Scheduler() decide when to call the Producer() or Consumer() functions. If the user wanted this to be done in a more methodical and predictable way, the program would have to be altered a bit more – however given the goal of the assignment this is to be expected.

Global variables are also a bad practice. Keeping to strict object-oriented principles, they shouldn’t be used as they aren’t objects. A solution to this would have been a sequenceValue class which is altered through .get() and .set() functions, which in a larger program is easier to keep track of where the value is being altered from. In this example though, and for its intended use, the global variable serves the purpose, and is of course still static.