Introduction

This assignment aims to implement a general purpose simulation program to determine the average waiting time of items such as jobs, persons, etc. This general purpose simulation program is a time-driven multiple server, single queue system which there are 5 servers in total and the items in the single queue will assign job in the queue to servers that are free. The process is carried out for six minutes and during the six minutes, jobs will reach at a random time. The job is then either assign to a server directly if the server is free or enqueue into the single queue if all the servers are busy. The arrival rate of jobs in this simulation program will be around 20 jobs per minute. The arrived job will be given a unique id to distinguish it from other jobs. The job is then given a service time which indicate the time for the job to be serviced. The service time for the jobs will be between 5 to 18 second. If the job is more than 8 seconds, for every 8 seconds, the job has to be enqueued back into the queue first then assign to other/same free server to continue job servicing.

Tools used

NetBeans IDE 8.1 is used to code for the program. Java programming language is used in coding for the program and the Graphical User Interface (GUI) of the program is created using Netbean IDE's built in GUI Builder.

Expected Outcome

This program should be able to serve arrived jobs within 6 minutes. It should be able to report the average waiting time for of jobs, total number of jobs processed in 6 minutes, average number of jobs processed per minute for each server, average arrival rate per minute, number of jobs processed on the first attempt, number of jobs had to be requeued once, number of jobs had to be requeued twice and total number of jobs processed by each server. During the 6 minutes program running time, the program should also be able to show jobs that are in the queue due to servers fully occupied and prioritize longer remaining duration jobs (allows longer remaining duration jobs to be serviced first).

Job class

Job class is a class that is a class which acts as a blueprint to create Job objects. Job objects contain all the information of the job, such as id, startTime, duration, duration and changingDuration. It implements comparable interface to implement the compareTo method which determines the priority of the job.

Data field

```
private final String id;
private int duration;
private int changingDuration;
```

Data Field	Data Type	Description
id	String	Store the job’s unique id
duration	Integer	Store the duration for the job to be serviced
changingDuration	Integer	Store the remaining duration for the job to be serviced (Change every second until finish service)

Job class constructor

```
public Job(String id, int duration) {
    this.id = id;
    this.duration = duration;
    this.changingDuration = duration;
}
```

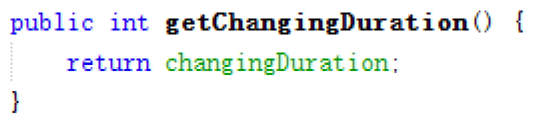
Job class constructor accepts two arguments, which are id and duration. The parameter id accepts unique id for the job and parameter duration accepts the duration for the job object. In the constructor body, the data field id is initialized with the id parameter argument and data field duration and changing duration is initialized with parameter duration.

decrement method

```
public void decrement() {
    --changingDuration;
}
```

In the Job class, decrement method contains only one statement which is pre-decrement changingDuration data field. This method will be called every second to decrease the remaining duration of the job to be serviced when it is servicing.

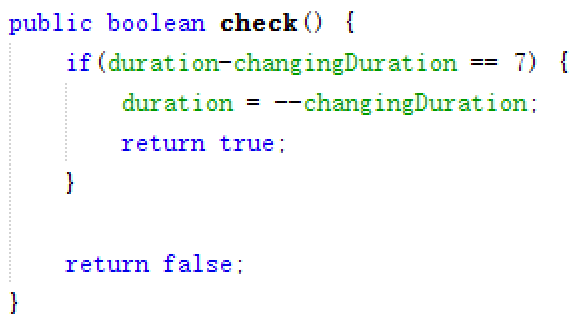
getChangingDuration method



```
public int getChangingDuration() {
    return changingDuration;
}
```

This accessor method returns the changingDuration data field to allow the remaining time to be checked in other classes.

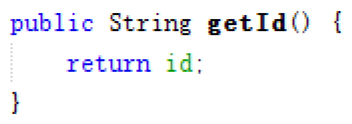
check method



```
public boolean check() {
    if(duration-changingDuration == 7) {
        duration = --changingDuration;
        return true;
    }

    return false;
}
```

This method decrease the changingDuration data field and assigned it to duration data field then return true if the subtraction of duration and changingDuration is 7. This is to make sure that the first second is also counted in the subtraction and the serviced time is exactly 8 seconds.

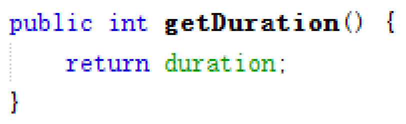
getId method



```
public String getId() {
    return id;
}
```

This accessor method returns the job’s unique id so that it can be accessed by other classes.

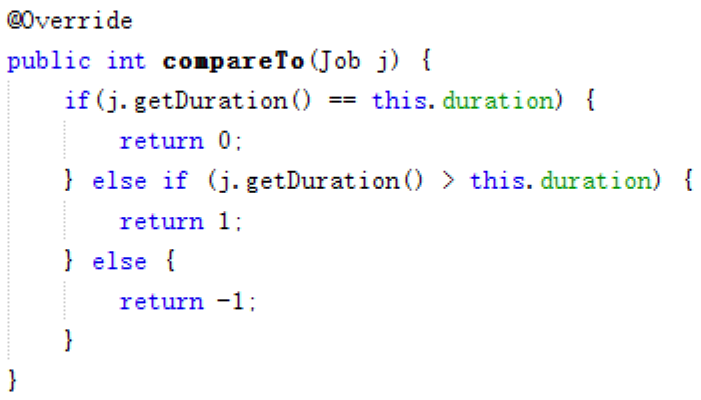
getDuration method



```
public int getDuration() {
    return duration;
}
```

This accessor method returns the job’s duration so that it can be accessed by other classes.

compareTo method



```java
@Override
public int compareTo(Job j) {
    if(j.getDuration() == this.duration) {
        return 0;
    } else if (j.getDuration() > this.duration) {
        return 1;
    } else {
        return -1;
    }
}
```

This method checks the duration of the job and return either 0, 1 or -1. As this system prioritize longer duration job, therefore, if the job has same duration as the compared job 0 is returned (no priority); if the compared job has longer duration, 1 is returned (lower priority); else -1 is returned, -1 is returned (higher priority).

Server Class

Server class is a class which contains properties and behavior of a server. It is a template used to create server objects. In this program, it is used to create five servers which is used to provide services on jobs.

Data Field

```java
private boolean free;
private Job job;
private int counter;
```

Data Field	Data Type	Description
free	Boolean	Indicate whether the server is free using true and false.
job	Job	Contains job that the server is currently servicing.
counter	Integer	Counter that is used to count the number of jobs processed by the server.

Server class constructor

```java
public Server() {
    this.free = true;
    counter = 0;
}
```

This constructor is a no-argument constructor. It is used to initialize the properties of the server, by initializing free instance variable to be true and counter to be 0. This is because when a new server object is created, the server should be free and the counter should be 0 as there is no job processed yet for new server object.'

isFree method

```java
public boolean isFree() {
    return free;
}
```

This method is used to return the free instance variable when invoked to check whether the server is free and to decide whether a new job should be assigned to it.

setJob method

```java
public void setJob(Job job) {
    this.job = job;
}
```

This method is used to assign new job to the server. When a server is free, if the queue is not empty, a new job will be passed into the server by assigning the job as argument and invoke this method.

decreaseTime method

```
public void decreaseTime() {
    job.decrement();
}
```

This method contains only one statement which is used to call decrement method of the job that the server is currently servicing. This will then decrease the changingDuration instance variable of the server which contains remaining duration for the job to be serviced.

setFree method

```
public void setFree(boolean free) {
    this.free = free;
}
```

This method is used to set the free instance variable of server object. It receives an argument which is a Boolean that indicates either true(server is free) and false(server is busy).

getJob method

```
public Job getJob() {
    return job;
}
```

This method is used to return the job that the server is currently servicing. This is to get the job that the server is servicing so that it's operation and behavior can be invoked and executed.

increaseCounter method

```
public void increaseCounter() {
    ++counter;
}
```

This method contains only one statement, which is to pre-increment the counter instance variable. This method will be called when a new job is assigned to the server so that it can increase the counter to count the total number of job serviced by the server.

getCounter method

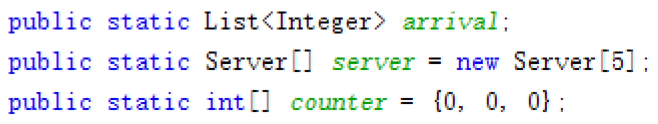
```
public int getCounter() {
    return counter;
}
```

This method returns the counter of the server. It is executed after the program runs for 6 minutes to get the total number of job serviced by the server and display it to user.

Task Class

This class can be considered the repository of the system. It contains many functions and data that are invoked and used during the 6 minutes of running time. It is a class that contains many data and methods that can be used without the need of creating a Task object as they are static variables and methods.

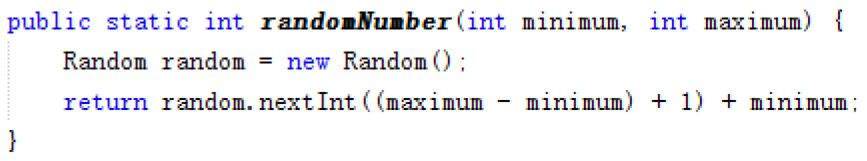
Data Field



```
public static List<Integer> arrival;
public static Server[] server = new Server[5];
public static int[] counter = {0, 0, 0};
```

Data Field	Data Type	Description
arrival	Integer List	Contains all the arrival time of the jobs within the 6 minutes.
server	Server Array	Contains server objects that are used to service the jobs.
counter	Integer Array	Counter that is used to count the number of jobs that do not need to requeue, need to requeue once and need to requeue twice.

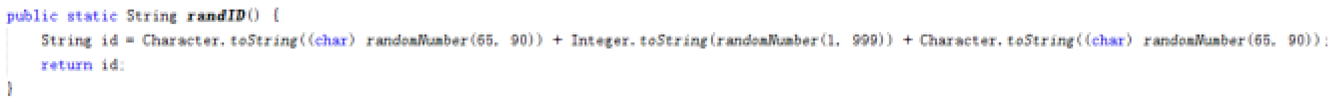
randomNumber method



```
public static int randomNumber(int minimum, int maximum) {
    Random random = new Random();
    return random.nextInt((maximum - minimum) + 1) + minimum;
}
```

This method receives two arguments, minimum and maximum. These two arguments represent the minimum range and maximum range of values to be randomly generated. By passing in these two variables, a Random instance is created and the randomly created number is returned.

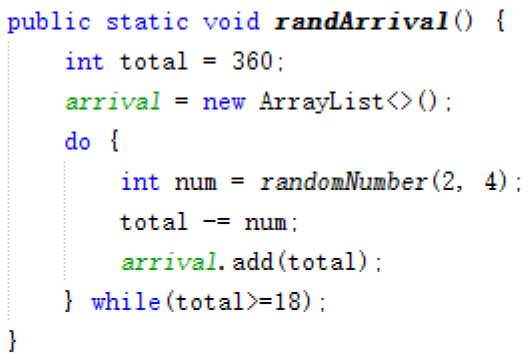
randID method



```
public static String randID() {
    String id = Character.toString((char) randomNumber(65, 90)) + Integer.toString(randomNumber(1, 999)) + Character.toString((char) randomNumber(65, 90));
    return id;
}
```

This method randomly generates two upper case character and a random number to form the id for job. The generated random id is then returned.

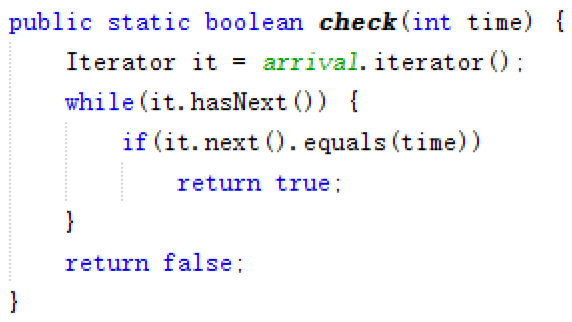
randArrival method



```
public static void randArrival() {
    int total = 360;
    arrival = new ArrayList<>();
    do {
        int num = randomNumber(2, 4);
        total -= num;
        arrival.add(total);
    } while(total>=18);
}
```

This method randomly generates the arrival time of the job within 6 minutes (360 seconds). However, a condition is used to ensure that the last 17 seconds will not have any job arrival. This is to ensure that there is enough time for the server to finish serving all the jobs and decrease the number of leftover customer which have not finish servicing after 6 minutes.

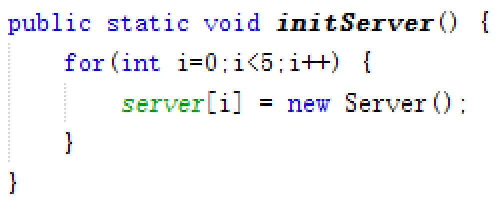
check method



```
public static boolean check(int time) {
    Iterator it = arrival.iterator();
    while(it.hasNext()) {
        if(it.next().equals(time))
            return true;
    }
    return false;
}
```

This method receives the current time of the system which is then compared with the arrival list. If the time passed in is found in the arrival list means that job have to be created. The method will then return true to indicate new job needs to be created.

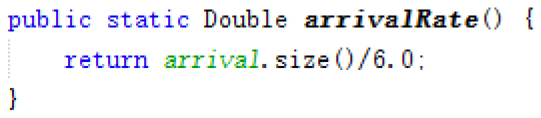
initServer method



```
public static void initServer() {
    for(int i=0;i<5;i++) {
        server[i] = new Server();
    }
}
```

This method is used to initialize all the servers in the server list so that it can be accessed and used later.

arrivalRate method



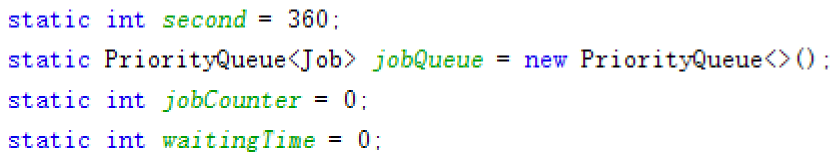
```
public static Double arrivalRate() {
    return arrival.size()/6.0;
}
```

This method returns the result of size of arrival list, which is actually the number job created in 6 minutes, divided by 6 (6 minutes running time).

DSAwithUI Class

This class is the main class which uses all the classes above to provide function and implements the system. It is the brain and interface of system which allows the system UI to be displayed to user. It extends javax.swing.JFrame so that it can be displayed in a Windows form.

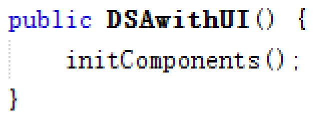
Variables declaration



```
static int second = 360;
static PriorityQueue<Job> jobQueue = new PriorityQueue<>();
static int jobCounter = 0;
static int waitingTime = 0;
```

Data Field	Data Type	Description
second	Integer	The number of seconds the system should run. Fix at 360 because the system will run for 6 minutes.
jobQueue	PriorityQueue<Job>	Contains jobs that are waiting to be processed. Priority Queue is implemented to prioritized Job with more remaining service time.
jobCounter	Integer	Count the number of Job arrived / created.
waitingTime	Integer	Count the total waiting time for all the jobs in the 6 minutes.

DSAwithUI Constructor

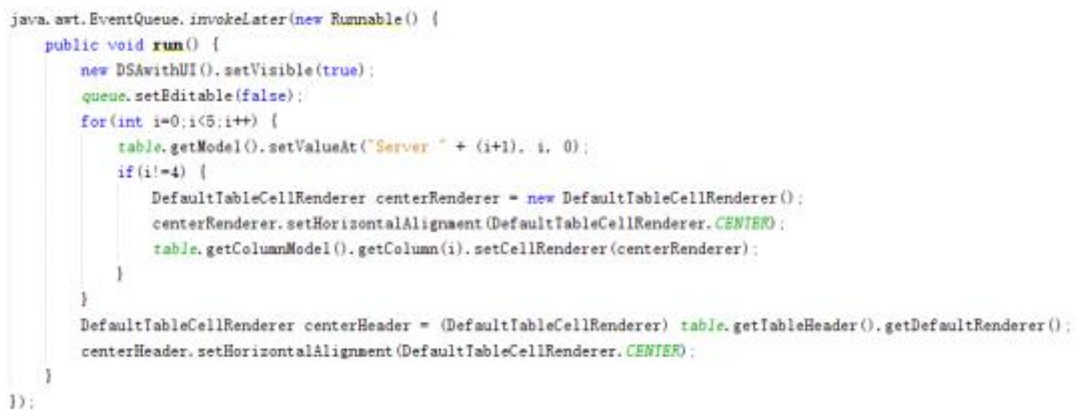


```
public DSAwithUI() {
    initComponents();
}
```

This constructor invoke initComponent method which is used to initialize all the components in the interface in the beginning of the system.

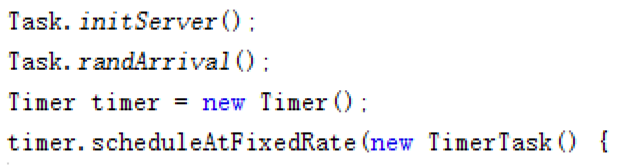
main method

This method contains all the codes and statements which will be executed to make the system works.



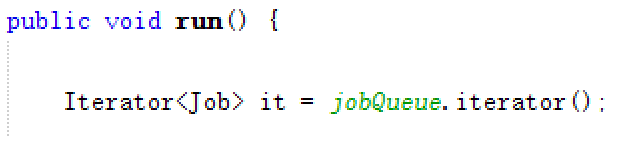
```
java.awt.EventQueue.invokeLater(new Runnable() {
    public void run() {
        new DSAwithUI().setVisible(true);
        queue.setEditable(false);
        for(int i=0;i<5;i++) {
            table.getModel().setValueAt("Server " + (i+1), i, 0);
            if(i!=4) {
                DefaultTableCellRenderer centerRenderer = new DefaultTableCellRenderer();
                centerRenderer.setHorizontalAlignment(DefaultTableCellRenderer.CENTER);
                table.getColumnModel().getColumn(i).setCellRenderer(centerRenderer);
            }
        }
        DefaultTableCellRenderer centerHeader = (DefaultTableCellRenderer) table.getTableHeader().getDefaultRenderer();
        centerHeader.setHorizontalAlignment(DefaultTableCellRenderer.CENTER);
    }
});
```

The statement above invokes invokeLater method in EventQueue and implement a new Runnable class so that the components of the user interface can be initialize and set up properly.



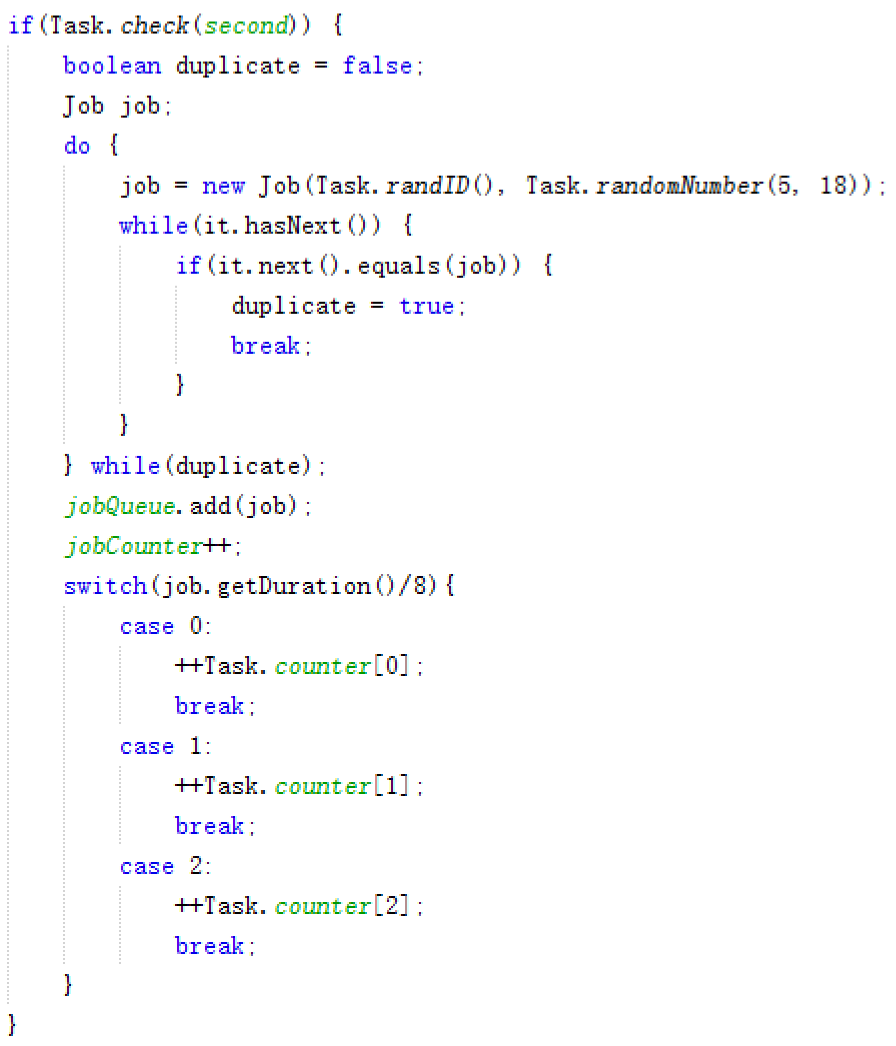
```
Task.initServer();
Task.randArrival();
Timer timer = new Timer();
timer.scheduleAtFixedRate(new TimerTask() {
```

These statements will invoke initServer method in Task class which can initialize all the Servers before they start using it. Then, randArrival method in Task class is invoked to randomly create time for the job to arrive for each 3 to 5 seconds. Then, a Timer object, timer, is instantiated so that is can be used to count the time remaining for execution later. scheduleAtFixedRate method is the invoked by timer object to time the system running time. A new TimerTask class implemented as the argument of scheduleAtFixedRate method to allow all the statements in the new TimerTask object to be executed every second. The implementation of TimerTask object will be as below.



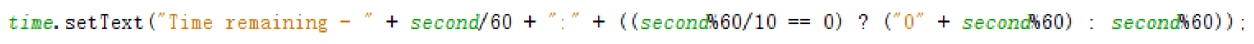
```
public void run() {

    Iterator<Job> it = jobQueue.iterator();
```

The TimerTask object’s run method contains all the statements that will be ran each second. An iterator object is created using iterator method of jobQueue so that it can be used more conveniently later when needs to iterate through jobQueue’s Job elements.



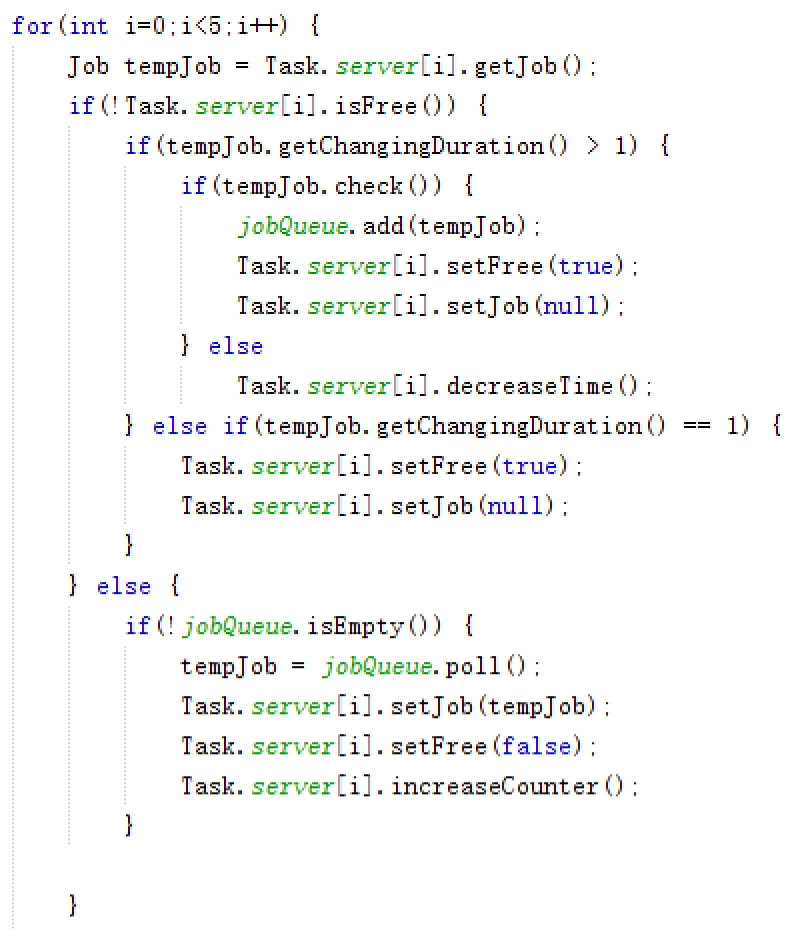
```
if(Task.check(second)) {
    boolean duplicate = false;
    Job job;
    do {
        job = new Job(Task.randID(), Task.randomNumber(5, 18));
        while(it.hasNext()) {
            if(it.next().equals(job)) {
                duplicate = true;
                break;
            }
        }
    } while(duplicate);
    jobQueue.add(job);
    jobCounter++;
    switch(job.getDuration()/8){
        case 0:
            ++Task.counter[0];
            break;
        case 1:
            ++Task.counter[1];
            break;
        case 2:
            ++Task.counter[2];
            break;
    }
}
```

The condition of this if statement is invoking the check method in Task class and passing second variable as argument to check whether a job should be created at this second. If check methos returns true, a do…while loop will be executed to generate a random Job object and make sure the new Job object is unique by ID. If it is not unique, the duplicate variable will be false and continue executing statements in do…while loop until a unique Job is created. After creating the new Job object, it will then be added to jobQueue and the jobCounter will be post-incremented. A switch statement is executed to check the number of times the new job needs to requeue.



```
time.setText("Time remaining - " + second/60 + ":" + ((second%60/10 == 0) ? ("0" + second%60) : second%60));
```

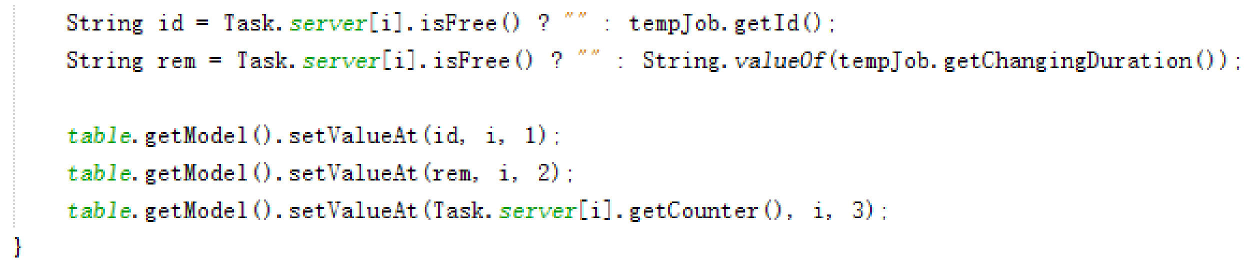
This statement will display the time remaining for the system to run in the user interface.



```
for(int i=0;i<5;i++) {
    Job tempJob = Task.server[i].getJob();
    if(!Task.server[i].isFree()) {
        if(tempJob.getChangingDuration() > 1) {
            if(tempJob.check()) {
                jobQueue.add(tempJob);
                Task.server[i].setFree(true);
                Task.server[i].setJob(null);
            } else
                Task.server[i].decreaseTime();
        } else if(tempJob.getChangingDuration() == 1) {
            Task.server[i].setFree(true);
            Task.server[i].setJob(null);
        }
    } else {
        if(!jobQueue.isEmpty()) {
            tempJob = jobQueue.poll();
            Task.server[i].setJob(tempJob);
            Task.server[i].setFree(false);
            Task.server[i].increaseCounter();
        }

    }
```

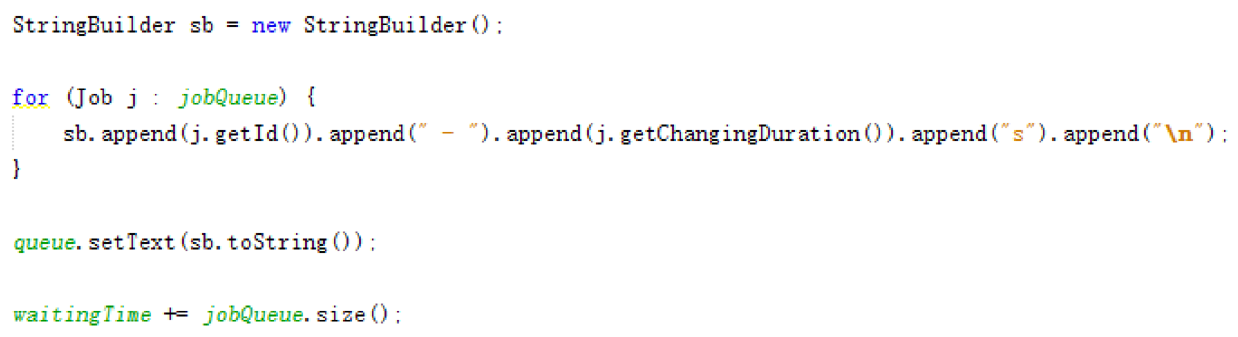
This for loop will make sure all Servers are kept busy in servicing Jobs. If the server is busy, the server will decrease the remaining time for the job it is currently servicing. If the job already serviced for 8 seconds, it will then be added into jobQueue and the server will be set to true. If the job is completed (remaining time equals to 1 because 0 is not considered a second), the server will be set free and null will be set to the job.

If the server is free, another nested if statement will check whether jobQueue is empty. If jobQueue is not empty, the prioritized job will be polled and set assigned it to the server. The server will then be set to busy by setting the free variable false.



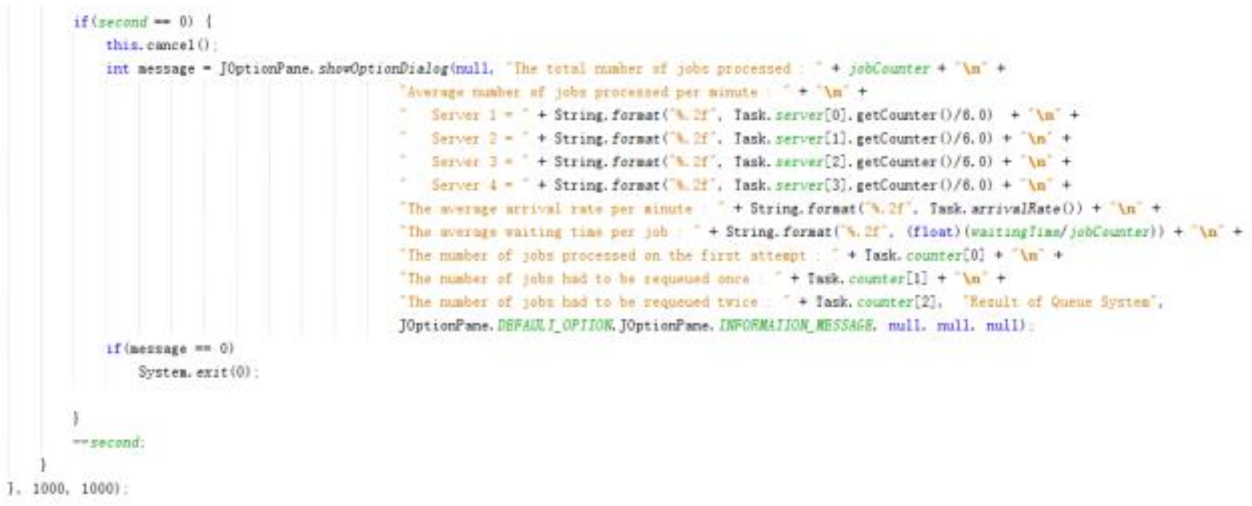
```
    String id = Task.server[i].isFree() ? "" : tempJob.getId();
    String rem = Task.server[i].isFree() ? "" : String.valueOf(tempJob.getChangingDuration());

    table.getModel().setValueAt(id, i, 1);
    table.getModel().setValueAt(rem, i, 2);
    table.getModel().setValueAt(Task.server[i].getCounter(), i, 3);
}
```

These statements used to update the value of the table which shows the job id which the server is currently servicing and the number of job serviced by server.



```
StringBuilder sb = new StringBuilder();

for (Job j : jobQueue) {
    sb.append(j.getId()).append(" - ").append(j.getChangingDuration()).append("s").append("\n");
}

queue.setText(sb.toString());

waitingTime += jobQueue.size();
```

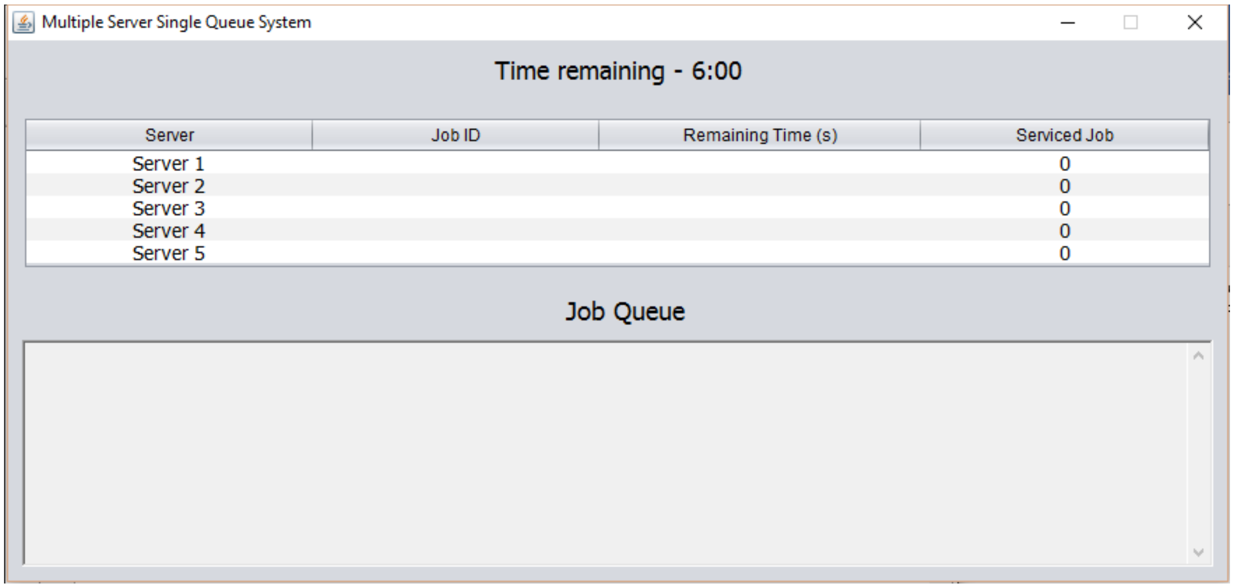
These statements use a string builder to build a string which contains all the jobs that are currently in the queue. After finish creating a string that contain all the jobs, the string is displayed in the TextArea so that the jobs that are currently queueing can be displayed to user. The last statement from the above picture is used to add the number of jobs in jobQueue to waitingTime variable. After the execution of most of the instructions which is related to the server above, the leftover jobs in the jobQueue will be waiting for the next second. Therefore, by adding the number of jobs in the jobQueue to waitingTime for each second, the total waiting time for all the jobs can be counted.



```
            if(second == 0) {
                this.cancel();
                int message = JOptionPane.showOptionDialog(null, "The total number of jobs processed : " + jobCounter + "\n" +
                                                "Average number of jobs processed per minute : " + "\n" +
                                                "    Server 1 = " + String.format("%.2f", Task.server[0].getCounter()/6.0)  + "\n" +
                                                "    Server 2 = " + String.format("%.2f", Task.server[1].getCounter()/6.0) + "\n" +
                                                "    Server 3 = " + String.format("%.2f", Task.server[2].getCounter()/6.0) + "\n" +
                                                "    Server 4 = " + String.format("%.2f", Task.server[3].getCounter()/6.0) + "\n" +
                                                "The average arrival rate per minute : " + String.format("%.2f", Task.arrivalRate()) + "\n" +
                                                "The average waiting time per job : " + String.format("%.2f", (float)(waitingTime/jobCounter)) + "\n" +
                                                "The number of jobs processed on the first attempt : " + Task.counter[0] + "\n" +
                                                "The number of jobs had to be requeued once : " + Task.counter[1] + "\n" +
                                                "The number of jobs had to be requeued twice : " + Task.counter[2], "Result of Queue System",
                                                JOptionPane.DEFAULT_OPTION, JOptionPane.INFORMATION_MESSAGE, null, null, null);

                if(message == 0)
                    System.exit(0);

            }
            --second;
        }
}, 1000, 1000);
```

When second equals to 0 which means 6 minutes reached, cancel method of the TimerTask object will be called to stop executing the instructions in the run method. An option dialog will be shown which will list out all the results. If “OK” button in the dialog is clicked, the system will then exit. The pre-decrement second variable is to decrease the time every second until the second reaches 0.

Screenshot of Multiple Server Single Queue System



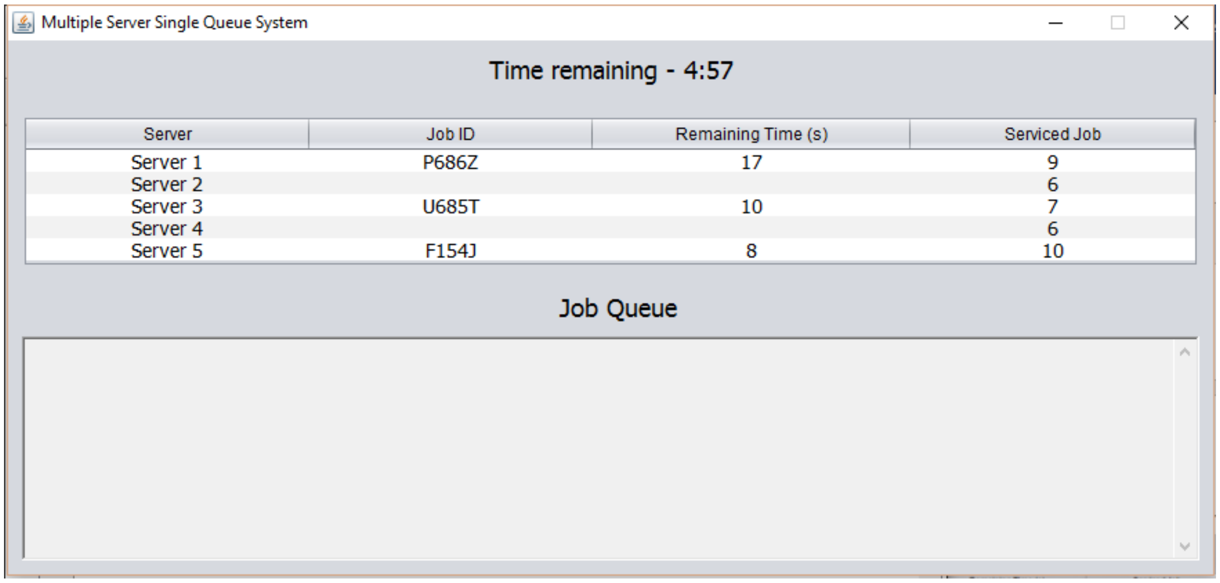
Multiple Server Single Queue System

Time remaining - 6:00

Server	Job ID	Remaining Time (s)	Serviced Job
Server 1			0
Server 2			0
Server 3			0
Server 4			0
Server 5			0

Job Queue

Figure 1 : Starting of System



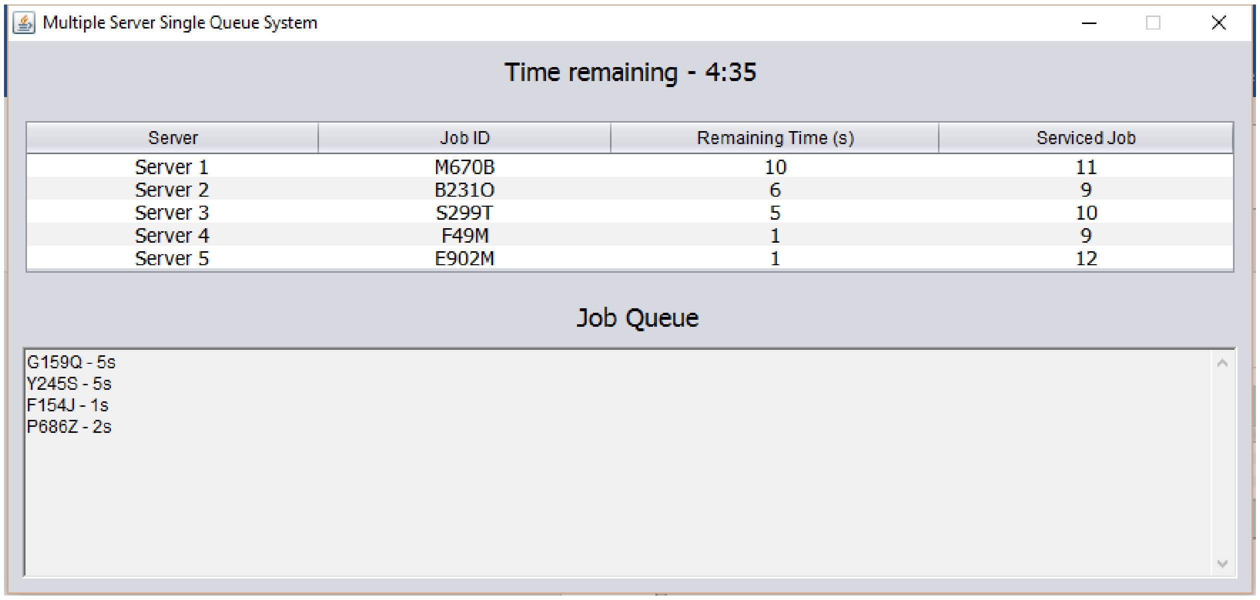
Multiple Server Single Queue System

Time remaining - 4:57

Server	Job ID	Remaining Time (s)	Serviced Job
Server 1	P686Z	17	9
Server 2			6
Server 3	U685T	10	7
Server 4			6
Server 5	F154J	8	10

Job Queue

Figure 2 : Servers are servicing Jobs



Multiple Server Single Queue System

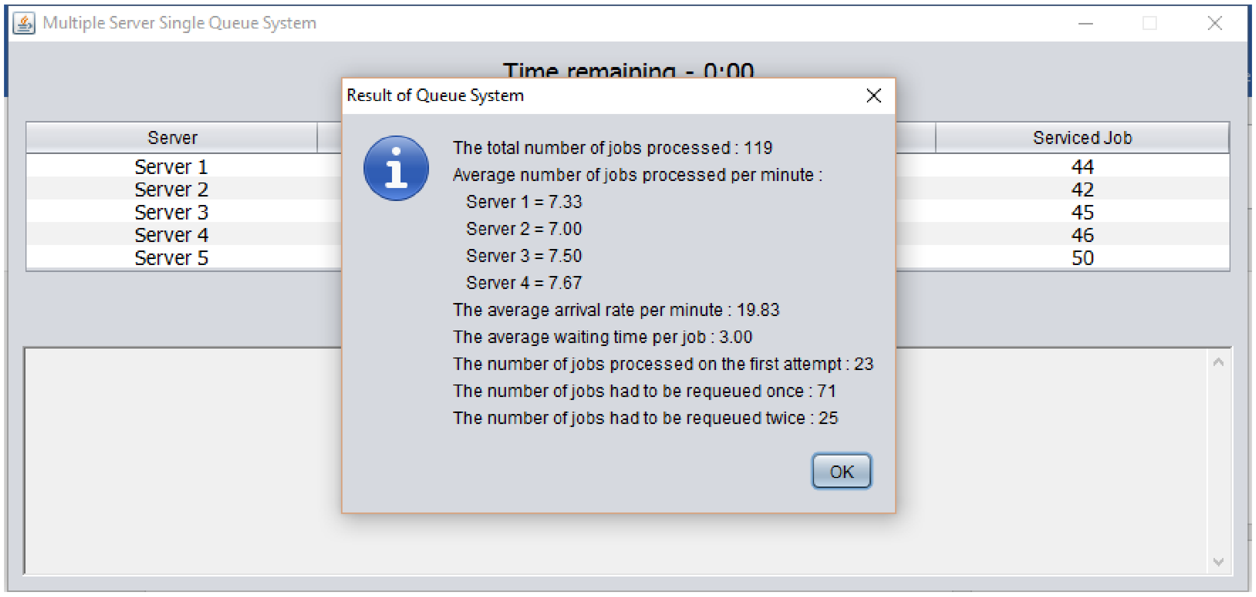
Time remaining - 4:35

Server	Job ID	Remaining Time (s)	Serviced Job
Server 1	M670B	10	11
Server 2	B231O	6	9
Server 3	S299T	5	10
Server 4	F49M	1	9
Server 5	E902M	1	12

Job Queue

```
G159Q - 5s
Y245S - 5s
F154J - 1s
P686Z - 2s
```

Figure 3 : Jobs in Job Queue when servers is full



Multiple Server Single Queue System

Time remaining - 0:00

Server	Serviced Job
Server 1	44
Server 2	42
Server 3	45
Server 4	46
Server 5	50

Result of Queue System

```
The total number of jobs processed : 119
Average number of jobs processed per minute :
   Server 1 = 7.33
   Server 2 = 7.00
   Server 3 = 7.50
   Server 4 = 7.67
The average arrival rate per minute : 19.83
The average waiting time per job : 3.00
The number of jobs processed on the first attempt : 23
The number of jobs had to be requeued once : 71
The number of jobs had to be requeued twice : 25
```

Figure 4 : End of system execution

Strength

The strength of this system is that it allows 5 servers to serve on a single queue. This decrease the burden of servers and allows jobs to be allocated to each servers equally. This also results in more jobs being served in 6 minutes compared to single server single queue system.

Weakness

The running time for this system is fixed at 6 minutes and the number of server is fixed at 5.

Conclusion

After testing and carry out validations on this multiple server single queue system, it can be said that objectives of this system were met. This system is created using Java.

Even though this program is successfully created, there is no perfect application. This program also contains some limitations. These limitations are, running time is fixed at 6 minutes and number of server is fixed at 5 servers.

This system can be improved by adding more functions. In this student survey application, it can be improved by allowing user to input the number of servers and the time for running the system.