```java
package Utilities;

import java.util.HashMap;


// RoundRobin class:
// This has some n processes and a quantum
public class RoundRobin {
    ArrayQueue<Process> processes;
    int quantum;
    int n;
    /*
     * Uses an ArrayQueue to construct processes such that their Burst time is given.
     */
    public RoundRobin(int n, int[] bTimes, int quantum) {
        this.n = n;
        this.processes = new ArrayQueue(n);
        this.quantum = quantum;
        for (int i = 0; i < n; i++) {
            this.processes.enqueue(new Process(i, bTimes[i], 0));
        }
    }

    public RoundRobin(int n, int[] bTimes, int[] aTimes, int quantum) {
        this.n = n;
        this.processes = new ArrayQueue(n);
        this.quantum = quantum;
        for (int i = 0; i < n; i++) {
            this.processes.enqueue(new Process(i, bTimes[i], aTimes[i]));
        }
    }
    /*
     * Average waiting Time is implemented using this method.
     * This method evaluates the average waiting time by manipulating the ArrayQueue.
     * It creates a gantt chart which is not stored in memory
     * However, that gantt chart is used to evaluate the time within the method only.
     * We use the methods enqueue and dequeue to traverese through next processes.
     *
     */
    public float averageWaitingTime() {
        int average = 0;
        int wait = 0;
        boolean loop = true;
        int time = 0;
        int secondaryTime = 0;
        int iterate = this.processes.size();
        int i = 0;

        while(this.processes.size()!=0) {//Loops until all the process finish
            i++;
            average += time; //waiting time for this process
            if(processes.first().timeLeft() < this.quantum) { //It evaluates to zero as it finishes the process before the quantum;
                this.processes.first().updateTimeLeft(this.processes.first().timeLeft());
                time  = this.processes.first().burstTime();
            }
            else {
                this.processes.first().updateTimeLeft(this.quantum); //evaluate the remaining time for this process
                time = this.quantum;
            }
            if (this.processes.first().timeLeft() != 0) {
```

```
61                  Process p = this.processes.dequeue();
62                  this.processes.enqueue(p); //this puts the process at the end of the line
63              }
64              else {
65                  this.processes.dequeue();
66                  secondaryTime += time;
67              }
68
69              if(loop ) {
70                  wait += average;
71              }
72              if(i == iterate && (this.processes.size() != 0)) {
73                  time = secondaryTime;
74                  wait += time;
75                  secondaryTime = 0;
76                  iterate = 0;
77                  loop = false;
78                  average = 0;
79              }
80
81          }
82          float avTime = ((float)wait)/this.n;
83          return avTime;
84      }
85  /*
86   * This is a dummy version of the gantt Chart
87   * It is not included in the code because for this calculation of the
88   * average time, we do not need the Gantt Chart as it is.
89   */
90  //  public HashMap<Integer, Process> ganttChart() {
91  //      HashMap<Integer, Process> gChart = new HashMap<>();
92  //      int time = 0;
93  //      int gTime  = 0;
94  //      while(this.processes.size()!=0) {
95  //          gTime += time;
96  //          if(processes.first().timeLeft() < this.quantum) { //It evaluates to zero as it
    finishes the process before the quantum;
97  //              this.processes.first().updateTimeLeft(this.processes.first().timeLeft());
98  //              time  = this.processes.first().burstTime();
99  //          }
100 //
101 //          else {
102 //              this.processes.first().updateTimeLeft(this.quantum); //evaluate the
    remaining time for this process
103 //              time = this.quantum;
104 //          }
105 //          gChart.put(time, this.processes.first());
106 //          if (this.processes.first().timeLeft() != 0) {
107 //              Process p = this.processes.dequeue();
108 //              this.processes.enqueue(p); //this puts the process at the end of the line
109 //          }
110 //          else {
111 //              this.processes.dequeue();
112 //          }
113 //
114 //
115 //      }
116 //      return gChart;
117 //  }
118
119 }
120 /*
```

```java
 * This is a Process object which have several attributes
 * and methods related to these attributes.
 */

class Process{
    private int id; //Process Id
    private int burstTime;// Burst time of a process
    private int waitTime;// waiting time for this process
    private int arrivalTime; // arrival time for this process
    private int timeLeft; // remaining time for each process

    public Process(int id, int bT, int at) { //Constructor for this object
        this.id = id;
        this.burstTime = bT;
        this.arrivalTime = at;
        this.timeLeft = this.burstTime;
    }

    public void addWaitTime(int n) { //Adds to the wait time some additional time 'n'
        this.waitTime += n;
    }

    public void updateTimeLeft(int n) { //Changes the time left to complete this process
        this.timeLeft -= n;
    }

    public int waitTime() { //returns the wait time for the process (int value)
        return this.waitTime;
    }

    public int burstTime() {// returns the integer value of Burst time
        return this.burstTime;
    }

    public int id() { //returns the Id of this process
        return this.id;
    }

    public int arrivalTime() { //returns the arrival time of the process
        return this.arrivalTime;
    }

    public int timeLeft() { //returns the time left to complete this process;
        return this.timeLeft;
    }
}


/*
 * FIFO implementation:
 * A class is implemented within the Round Robin class to use this class
 * for the implementation. This is just an ADT of the  type ArrayQueue.
 */
class ArrayQueue<E>{
    static int CAPACITY = 100;
    private E[] data;
    private int front = 0;
    private int size = 0;

    public ArrayQueue() {
        this(CAPACITY);
    }
```

```
183
184     public ArrayQueue(int capacity) {
185         try {
186             data = (E[]) new Object[capacity];
187         } catch (Exception e) {
188             System.out.println(e.getMessage());
189         }
190     }
191
192     public int size() {
193         return this.size;
194     }
195
196     public boolean isEmpty() {
197         return (this.size == 0);
198     }
199
200     public void enqueue(E e) throws IllegalStateException{
201         if (this.size == this.data.length) throw new IllegalStateException("Queue is
    full");
202         int avail = (this.front + this.size) % this.data.length;
203         this.data[avail] = e;
204         this.size++;
205     }
206
207     public E first(){
208         if(isEmpty()) return null;
209         return this.data[this.front];
210     }
211
212     public E dequeue() {
213         if(isEmpty()) return null;
214         E answer = this.data[this.front];
215         this.data[this.front] = null;
216         this.front = (this.front + 1) % this.data.length;
217         this.size--;
218         return answer;
219     }
220
221
222 }
```