Practical Assignment 3

- ☐ **Date Issued:** Monday, 24 August 2020
- ☐ **Date Due:** Sunday, 06 September 2020, 18:00
- ☐ **Submission Procedure:** Upload your tasks on the CS Assignments Portal website
- ☐ **Assessment:** The practical will be marked online through the Assignments Portal and slots will open in the second week of the practical.
- ☐ This assignment consists of 2 tasks

---

1 Introduction

1.1 Objectives and Outcomes

You must complete this assignment individually.

You may ask the Teaching Assistants for help but they will not be able to give you the solutions. They will be able to help you with coding, debugging and understanding the concepts explained both in the textbook and during lectures.

1.2 Source Code

Before you begin, you should download the source code from the ClickUP website. You are allowed to make changes to all the files in the source code. Ensure that your **student number** and **name** appears in the comments section at the top of each file that you submit.

1.3 Mark Allocation

This assignment is divided into two tasks. For each task:

(a) your program must produce the expected output;

(b) your program must not throw any exceptions; and

(c) your program must implement the correct mechanisms or algorithms, as specified by the assignment.

Your code will be assessed online through a script. Please note that the same marking rubric is used for auto-assessment and manual marking. If you do wish to have the structure of you code marked offline you can query after you receive your marks.

The marks for this practical are allocated as follows:

Activity	Mark
Task 1	12
Task 2	14
Total	**26**

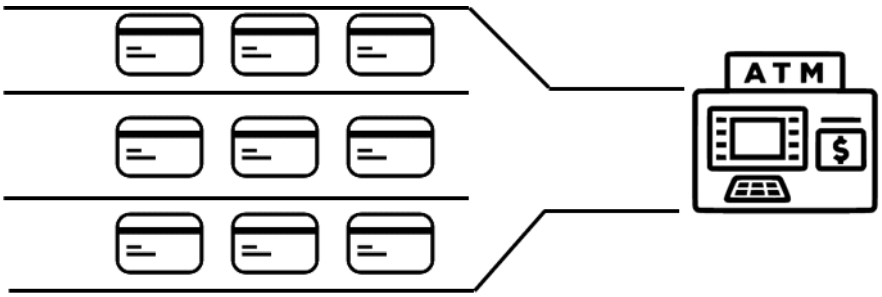
2 Implementation

By now you have should know the implementation of a few different types of locking methods.
Tips for implementation:

- For this practical you will be implementing the `Filter Lock` that you have learnt in class.
- If you are not sure where to start, take a look at the previous practicals, specifically at how everything fits in and think about the following:
  - The critical section is the part of a program that multiple threads should not access at the same time. After reading the scenario think about what that part is going to be?
  - In the previous practicals, multiple threads tried to acquire access to the critical section at the same time. How was this represented in the previous practicals and how can it be represented here?

2.1 Scenario - Single Line ATM Simulator

For this practical you need to imagine an ATM machine that allows users to withdraw and deposit money into it, similar to the banking account in the last practical. But only one credit card user can access the ATM machine at a time.
Consider the image below as an example:



2.2 Task 1 - Filter Lock Single Line ATM Simulator (10 marks)

Write a program implementation from scratch that will ensure that only one credit card user from a line uses the ATM at a time. `Hint:` your program should contain at least the following classes and functions per class:

- Line.java
  - Line(int numLines, ATM atm)
  - void setLines(cardUser[] lines)
  - cardUser[] getLines()

- CardUser.java
  - CardUser(ATM atm)
  - void setATM(ATM atm)
  - ATM getATM()
  - void run()

- ATM.java
  - ATM(float amount,int numThreads)
  - boolean withdraw(float amount,int threadID)
  - void deposit(float amount,int threadID)
  - float getATMBalance()

- FilterLock.java
  - FilterLock(int levels)
  - void lock()
  - void unlock()
  - boolean tryLock()

  - `boolean tryLock(long time, TimeUnit unit) throws InterruptedException`
  - `Condition newCondition()`
  - `void lockInterruptibly() throws InterruptedException`

- `ThreadATMDemo.java`
  - `static void main (String args[])`

The only files you will need to reuse is the ThreadID.java and logging.properties.
You may implement your own helper functions but be aware we will only be using the functions defined above.

Take the following into consideration:

- In order to test your code you will need to simulate the multiple lines and credit card users at the same time. In order to do this you will need to implement a class that uses threads in a similar that way you did in the last assignment. You will need to implement it in a way such that there are n number of lines, with c number of credit card users per line. Each transaction will need to sleep for *t1* and *t2* milliseconds representing using the ATM as well as representing a deposit or withdrawal respectively. The aforementioned values can be specified through the following `TransactionOptions` static variables:
  - n: `TransactionOptions.lines`
  - c: `TransactionOptions.cardUsers`
  - *t1*: `TransactionOptions.withdrawalSleepTime`
  - *t2*: `TransactionOptions.depositSleepTime`
- Please do not hardcode any values as our marking scripts tests with a unique set of values so all values used need to be done in a dynamic nature.
- For now you can test with 3 lines but create the constructor to grow and shrink the lines as needed.
- Each line contains a certain number of credit card users. Again this number can grow and shrink based on what is passed through to the constructor so plan accordingly. Test with 4 credit card users per line for now.
- Ensure that a credit card user only uses the ATM when there is no one else using it. This needs to be monitored via output.
- Whenever a credit card user uses the ATM they will either withdraw or deposit a random amount of money. The transaction they perform will also need to be picked at random between either a withdrawal or deposit.

- If they make a withdrawal then they need to withdraw a random amount of money ranging from R200,00 up to R1000,00.
- If there is not enough they will not make any transaction.
- If they make a deposit then they will deposit a random amount of money ranging from R50,00 to R500,00.

- Remember to keep track of amount of money left in the ATM after every transaction.

Implementation notes:

- Before a thread attempts to acquire the lock AND in between attempts, it should wait for a random amount of time.
  The following order needs to occur per thread:
  sleep ... access attempt ... sleep ... access attempt, etc.

- Remember multiple threads will try to access the critical section at the same time. Once a thread attempts to acquire the lock/synchronized block it should give output as follows:
  `Thread-$Thread-ID$ waiting to perform transaction`

  This shows that this thread is now competing for the lock and will have to wait its turn.

- Once a thread has gained access to the critical section it should output the following to show which thread is entering the critical section and which threads if any are still victims:
  `Thread-$Thread-ID$ performing transaction on the ATM(entering CS)`
  `Thread-$Victim-ID$ is a victim`
  `...`

- Depending on the transaction the credit card user wants to perform it can take two different amounts of time to perform that transaction. You will need to simulate this in your code by sleeping the threads for specified amounts of time given when the program starts.
  - For testing purposes only, if a thread is withdrawing money they can simulate waiting by sleeping for 150ms.
  - Again for testing purposes only, if a thread is depositing money they can simulate waiting by sleeping for 300ms.

- Once a thread has completed it's transaction on the ATM, your program should output: `Thread-$Thread-ID$ exiting`

- **Please note:** Threads can enter the critical section in any given order.

2.2.1 Program Logger

The program output will be displayed using a Logger object from the java.util.logging package. Each class that needs to display an output needs to declare a global Logger variable. For example:

```
private static final Logger LOGGER = Logger.getLogger("global");
```

The display configurations are specified in the logger.properties file.
This practical assignment will make use of 4 logging levels to display different types of messages. The logging levels in descending order are: `INFO, FINE, FINER, and FINEST.` When you set a level, all messages belonging to that level and higher levels will be displayed. For example, if the level is set to `FINER`, messages logged as `FINER, FINE`, and INFO will be displayed. Please make sure to implement your output messages using the **correct logging method**. Failure to do so may result in you not being awarded marks for certain items, as the marking script will use the logging levels to examine different types of thread behaviours.

Your program should produce the following outputs. The keywords enclosed in dollar signs **($keyword)** should be replaced with the associated data.

- For each ATM withdrawal, display a message that contains the name of the thread associated with a transaction, the transaction amount, and the remaining account balance . All Account messages should use the INFO logging level. Use the following format:

  ```
  Thread-$Thread-ID$ withdrawing R$amount$ from ATM, R$balance$ remaining
  ```

  $Thread-ID$ is the value returned from the ThreadID class that you used in the previous assignment.
  In the case of insufficient funds, the following message should be displayed:

  ```
  Thread-$Thread-ID$ withdrawal of R$amount$ not possible, insufficient
  funds.  R$balance$ remaining
  ```

- Log each deposit transaction as follows

  ```
  Thread-$Thread-ID$ depositing R$amount$ to ATM, R$balance$ remaining
  ```

- To analyze thread behaviour under the Filter lock algorithm, a message should be displayed when a thread becomes the victim, and when it is about to enter or leave the critical section (CS). The messages should be logged as FINE messages. The variable $thread$ shown below refers to the local thread ID generated by the lock.

  ```
  Thread-$Thread-ID$ waiting to perform transaction
  Thread-$Thread-ID$ performing transaction on the ATM(entering CS)
  Thread-$Victim-ID$ is a victim
  Thread-$Thread-ID$ exiting
  ```

Your output will follow the format below:

```
Thread-1 waiting to perform transaction
Thread-2 waiting to perform transaction
Thread-3 waiting to perform transaction
Thread-1 performing transaction on the ATM(entering CS)
Thread-2 is a victim
Thread-3 is a victim
Thread-1 depositing R66,00 to ATM. R5066,00 remaining
Thread-1 exiting
Thread-2 performing transaction on the ATM(entering CS)
Thread-3 is a victim
Thread-2 withdrawing R420,00 from ATM. R4646,00 remaining
Thread-2 exiting
...
Thread-2 performing transaction on ATM
Thread-2 withdrawal of R226,62 not possible, insufficient funds.  R182,00 remaining
Thread-2 exiting
```

Your output will be correct when:

- Your system is deadlock and starvation free – when a thread wants to perform a transaction on the ATM, it should eventually do so.
- There should be mutual exclusion on access to the ATM.
    - An easy way to see this is to make sure that in your output, one thread is not `performing transaction` on the `ATM` while another has not yet reached `exiting`.

Upload Instructions:
Compress all your files into a single archive and upload the archive to the Assignments portal website to the Practical 3 Task1 submission box. Make sure that the uploaded file does not belong to any package. **Assignments Portal**.

2.3 Task 2 - Fair Single Line ATM Simulator (12 marks)

When looking at the output from Task 1, you will notice that while the ATM is constantly used and no more than one credit card user uses the ATM at a time, some lines moves faster than others.

Please note you must still use the filter lock in this task.

For example consider the following possible output while running task 1:

```
Thread-1 waiting to perform transaction
Thread-2 waiting to perform transaction
Thread-3 waiting to perform transaction
Thread-1 performing transaction on the ATM(entering CS)
```

```
Thread-2 is a victim
Thread-3 is a victim
Thread-1 depositing R66,00 to ATM. R5066,00 remaining
Thread-1 exiting
Thread-2 performing transaction on the ATM(entering CS)
Thread-3 is a victim
Thread-2 withdrawing R420,00 from ATM. R4646,00 remaining
Thread-1 waiting to perform transaction
Thread-2 exiting
Thread-1 performing transaction on the ATM(entering CS)
Thread-3 is a victim
Thread-1 withdrawing R500,00 from ATM. R4146,00 remaining
Thread-1 exiting
Thread-2 waiting to perform transaction
Thread-3 performing transaction on the ATM(entering CS)
Thread-1 waiting to perform transaction
Thread-1 is a victim
Thread-2 is a victim
Thread-3 depositing R200,00 to ATM. R4346,00 remaining
Thread-3 exiting
```

In this example, `Thread-1` got to access the critical section twice, while `Thread-3` has not yet had an opportunity although it attempted to acquire the lock the same time as `Thread-1`. The current system is thus not first-come-first-served fair.

For this task, you are required to modify your Single Line ATM Simulator from Task 1 in such a way that it offers fairness.

Specifically, each line should only allow one credit card user to use the ATM and should then wait until all the other lines have each also allowed one credit card user to use the ATM. Once all the lines have had an opportunity, the next set of credit card users may enter.

Line tracking and output:
In order to ensure that fairness is permitted you will implement an array to keep track of the lines which have entered the critical section and the lines that have not. This needs to be a dynamic array that can grow and shrink in size depending on how many lines are used in the program.

- When the program starts, the array values should all be set to x to show that no threads have used the critical section. You will display this using an `[x]` to show it has not yet done anything.
  For example:
  ```
  Lines that used the ATM:
  [x]
  [x]
  [x]
  Thread-1 waiting to perform transaction
  ```

```
Thread-2 waiting to perform transaction
Thread-3 waiting to perform transaction
...
```

- When a thread enters and leaves the critical section it will update the array to specify that the specific line has gone by setting its index value to o. Before releasing the lock it will also output the current state of the array to show which lines have gone and which lines have not. All lines that have gone will be shown with a `[o]`. For example:

```
Lines that used the ATM:
[x]
[x]
[x]
Thread-1 waiting to perform transaction
Thread-2 waiting to perform transaction
Thread-3 waiting to perform transaction
Thread-1 performing transaction on the ATM(entering CS)
Thread-2 is a victim
Thread-3 is a victim
Thread-1 depositing R66,00 to ATM. R5066,00 remaining
Thread-1 exiting
Lines that used the ATM:
[o]
[x]
[x]
Thread-2 performing transaction on ATM(entering CS) victims:
Thread-3 is a victim
Thread-2 withdrawing R420,00 from ATM. R4646,00 remaining
Thread-1 waiting to perform transaction
Thread-2 exiting
Lines that used the ATM:
[o]
[o]
[x]
Thread-3 performing transaction on ATM(entering CS) victims:
no victim
Thread-3 depositing R200,00 to ATM. R4446,00 remaining
Thread-3 exiting
Lines that used the ATM:
[o]
[o]
[o]
...
```

- Once all lines have gone the array will reset itself to allow a new set of credit card users to proceed.
  For example:

```
...
```

```
Lines that used the ATM:
[o]
[o]
[o]
All lines used the ATM, resetting:
[x]
[x]
[x]
```

You may test with the same simulation environment as used for Task 1 but again please do not hard code these values:

- 3 lines
- 4 credit card users per line
- threads should sleep for a random amount of time in between attempts AND
- threads should sleep for the same time as specified in task 1 to simulate transaction time

Logger output
For the new addition to the output the format to follow on the logger is given below:

```
Lines that used the ATM:
[$array[$index 0$]$]
[$array[$index 1$]$]
...
[$array[$index n$]$]
```

To avoid any ambiguity in the above line of output please take note of the following:

- Think about how you can loop through the array and push the values to the logger.
- Update and display the line values just before you release the lock to ensure consistency and reset the array if needs be before another line sends a user to use the ATM.

Upload Instructions:
Compress all your files into a single archive and upload the archive to the Assignments portal website to the Practical 3 Task2 submission box. Make sure that the uploaded file does not belong to any package. **Assignments Portal**.

Upload slot release date:
Please note that the upload slots will open in the second week of the practical. Test slots may open before hand to allow you to test your code.