CA670 Concurrent Programming

Name	Aruna Bellgutte Ramesh
Student Number	18210858
Programme	MCM (Cloud Computing)
Module Code	CA670
Assignment Title	Java Threads
Submission date	18th March 2019
Module coordinator	David Sinclair

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

I have read and understood the referencing guidelines found recommended in the assignment guidelines.

Name: Aruna Bellgutte Ramesh Date: 18th March 2019

Java Threads

Architecture of the program

The program is constructed such that all the requirements mentioned in the assignment are satisfied. The project imitates the scenario of a company containing 10 departments, making transactions over 50 of its internal accounts. It also uses **MySQL database (MySQL DB)** to store and retrieve details of these 10 departments and 50 accounts. The Java program consists of seven classes. Each of these seven classes have specific functionality to do. Like:

1. **Account Class**: Imitates internal accounts of a company. Account is uniquely identified by **account_id** and has a balance held by **account_balance** variable. One can perform a deposit, withdrawal and/or transfer funds function on these accounts.
2. **AccountList Class**: Creates and returns an ArrayList of 50 accounts initialized with IDs from 1 to 50 and the balance of each of these 50 accounts is fetched from MySQL DB.
3. **Department Class**: Since according to the assignment, Department is the one that makes the transactions on the accounts, I have designed Department to implement the Runnable interface so as to create tasks (here transactions on accounts) for the threads to execute.
4. **DataAccess Class**: Implements the logic for connection to MySQL DB.
5. **Services Class:** Has all the Data Access Layer functions i.e., all the functions that interact with DB like getting balance for an account, updating balance for an account, getting account IDs and getting department IDs are written here.
6. **RandomGenerator Class:** This class has functions that returns a random Account ID from list of Account IDs present in DB, random Department ID from list of Department IDs present in DB, random amount (between the numbers 500 and 100) and random Transaction ID (between 0 - deposit, 1 – withdrawal and 2 – transfer funds) when requested.
7. **Main Class:** Contains the main function. **ExecutorService** and **Executors** are used to create and maintain a **thread pool**, to be run on **multiple cores**. Loops a 10,000 times and calls the RandomGenerator functions to get random account ID, department ID, transaction ID and amount. Using these values, tasks are created and submitted for threads to execute.

MySQL DB is used to store and retrieve details of accounts and departments. The initial data was created in a .csv file and imported into the table structure created in DB. The related SQL queries and .csv files are attached along with the submission folder.

Justifications

1. **Absence of thread starvation and fairness**
   One can confidently say that a program doesn't starve / choke its threads if the threads are well managed. In my program, I have used the **ExecutorService** and **Executors** interface which manages the threads created fair and square. It queues the tasks that are yet to be executed until at least one of its threads become free. The thread that becomes free first takes up the next task in the queue. This ensures **absence of thread starvation** and **fairness**.

2. **Absence of deadlocks**
   I have used **synchronized methods** to ensure that there is **no deadlocks**. Upon arrival of a critical section, synchronized methods helps a thread access the **shared resources** of a **critical section**, **synchronously**, ensuring other threads don't gain access to the critical section until the first thread leaves the said critical section. This way, there will **never** be a situation when two or more threads block each other.

3. **Correctness**
   We can say that a program is correct if it is free of both dead locks and thread starvation. Through points 1 and 2, I can also conclude that my program is **correct**.

Test Cases

1. Test Case 01:

Aim:

- To show that the program I have submitted is **void** of any race conditions, or deadlocks, or thread starvation.
- Demonstrate **deposit** functionality.

Test conditions:

Number of Cores: 4

Number of tasks (load): 10,000

Scenario: Keep depositing an amount of €20 to Account ID 2, whose initial balance is 0. Run this task a 10,000 times.

Expected Output: Final balance in Account ID 2 should be €2,00,000.

Actual Output: Final balance in Account ID 2 is €2,00,000.

Result: Pass

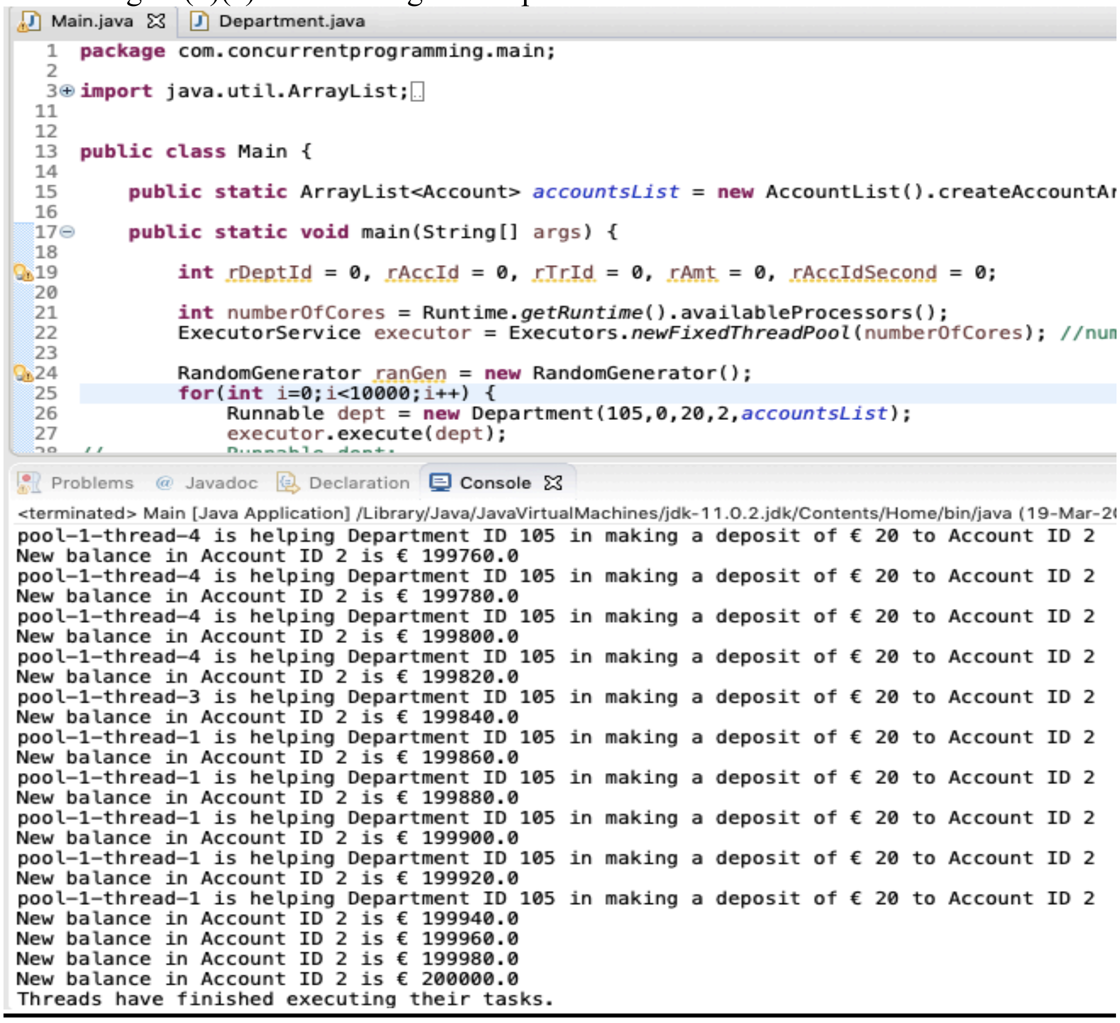
Conclusion:

- Program is void of race conditions, deadlocks and thread starvation i.e., the program is **correct**.
- **Deposit** functionality works as expected.

Screenshot: In figure (1)(a), (1)(b), (1)(c).

NOTE: Other screenshots and output of this testcase is submitted along with the submission folder.

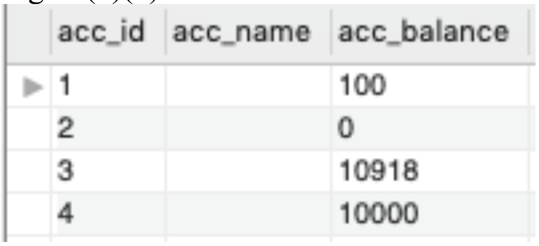
Figure(1)(a): TC01: Program output on CLI.



```
package com.concurrentprogramming.main;

import java.util.ArrayList;


public class Main {

    public static ArrayList<Account> accountsList = new AccountList().createAccountAr

    public static void main(String[] args) {

        int rDeptId = 0, rAccId = 0, rTrId = 0, rAmt = 0, rAccIdSecond = 0;

        int numberOfCores = Runtime.getRuntime().availableProcessors();
        ExecutorService executor = Executors.newFixedThreadPool(numberOfCores); //num

        RandomGenerator ranGen = new RandomGenerator();
        for(int i=0;i<10000;i++) {
            Runnable dept = new Department(105,0,20,2,accountsList);
            executor.execute(dept);
```

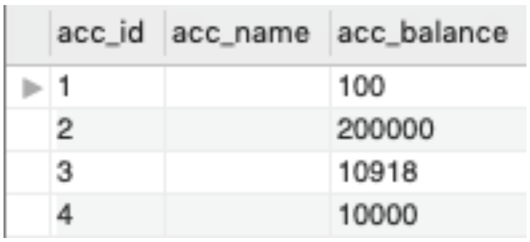
```
<terminated> Main [Java Application] /Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/bin/java (19-Mar-2
pool-1-thread-4 is helping Department ID 105 in making a deposit of € 20 to Account ID 2
New balance in Account ID 2 is € 199760.0
pool-1-thread-4 is helping Department ID 105 in making a deposit of € 20 to Account ID 2
New balance in Account ID 2 is € 199780.0
pool-1-thread-4 is helping Department ID 105 in making a deposit of € 20 to Account ID 2
New balance in Account ID 2 is € 199800.0
pool-1-thread-4 is helping Department ID 105 in making a deposit of € 20 to Account ID 2
New balance in Account ID 2 is € 199820.0
pool-1-thread-3 is helping Department ID 105 in making a deposit of € 20 to Account ID 2
New balance in Account ID 2 is € 199840.0
pool-1-thread-1 is helping Department ID 105 in making a deposit of € 20 to Account ID 2
New balance in Account ID 2 is € 199860.0
pool-1-thread-1 is helping Department ID 105 in making a deposit of € 20 to Account ID 2
New balance in Account ID 2 is € 199880.0
pool-1-thread-1 is helping Department ID 105 in making a deposit of € 20 to Account ID 2
New balance in Account ID 2 is € 199900.0
pool-1-thread-1 is helping Department ID 105 in making a deposit of € 20 to Account ID 2
New balance in Account ID 2 is € 199920.0
pool-1-thread-1 is helping Department ID 105 in making a deposit of € 20 to Account ID 2
New balance in Account ID 2 is € 199940.0
New balance in Account ID 2 is € 199960.0
New balance in Account ID 2 is € 199980.0
New balance in Account ID 2 is € 200000.0
Threads have finished executing their tasks.
```

Figure(1)(b): TC01: DB before.



acc_id	acc_name	acc_balance
1		100
2		0
3		10918
4		10000

Figure(1)(c): TC01: DB after.



acc_id	acc_name	acc_balance
1		100
2		200000
3		10918
4		10000

2. Test Case 02:

Aim:

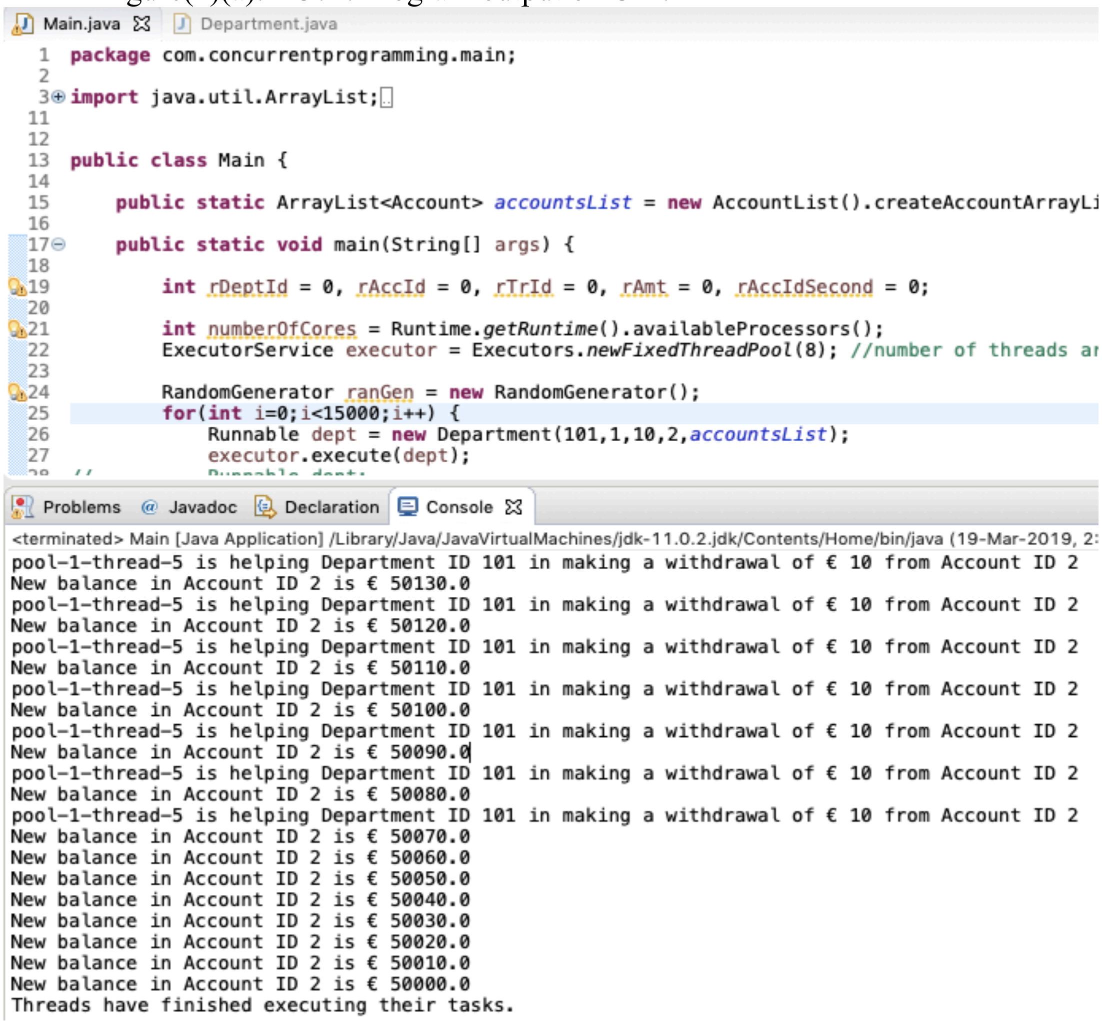
- Demonstrate variations in load and core.
- Demonstrate **withdrawal** functionality.

Test conditions:
Number of Cores: 8
Number of tasks (load): 15,000
Scenario: Now that the balance in Account ID 2 is €2,00,000; keep withdrawing an amount of €10 from Account ID 2. Run this task a 15,000 times.
Expected Output: Final balance in Account ID 2 should be €50,000.
Actual Output: Final balance in Account ID 2 is €50,000.
Result: **Pass**
Conclusion:

- Program is **correct** even for a load of 15,000 and 8 cores.
- **Withdrawal** functionality works as expected.

Screenshot: In figure (2)(a), (2)(b), (2)(c).
NOTE: Other screenshots and output of this testcase is submitted along with the submission folder.

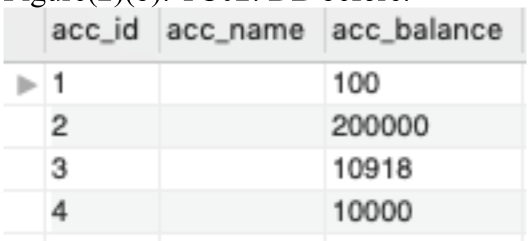
Figure(2)(a): TC02: Program output on CLI.



```java
package com.concurrentprogramming.main;

import java.util.ArrayList;


public class Main {

    public static ArrayList<Account> accountsList = new AccountList().createAccountArrayLi

    public static void main(String[] args) {

        int rDeptId = 0, rAccId = 0, rTrId = 0, rAmt = 0, rAccIdSecond = 0;

        int numberOfCores = Runtime.getRuntime().availableProcessors();
        ExecutorService executor = Executors.newFixedThreadPool(8); //number of threads ar

        RandomGenerator ranGen = new RandomGenerator();
        for(int i=0;i<15000;i++) {
            Runnable dept = new Department(101,1,10,2,accountsList);
            executor.execute(dept);
```

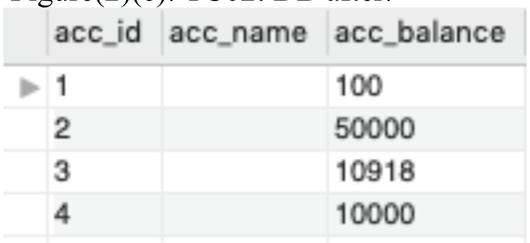
```
<terminated> Main [Java Application] /Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/bin/java (19-Mar-2019, 2:
pool-1-thread-5 is helping Department ID 101 in making a withdrawal of € 10 from Account ID 2
New balance in Account ID 2 is € 50130.0
pool-1-thread-5 is helping Department ID 101 in making a withdrawal of € 10 from Account ID 2
New balance in Account ID 2 is € 50120.0
pool-1-thread-5 is helping Department ID 101 in making a withdrawal of € 10 from Account ID 2
New balance in Account ID 2 is € 50110.0
pool-1-thread-5 is helping Department ID 101 in making a withdrawal of € 10 from Account ID 2
New balance in Account ID 2 is € 50100.0
pool-1-thread-5 is helping Department ID 101 in making a withdrawal of € 10 from Account ID 2
New balance in Account ID 2 is € 50090.0
pool-1-thread-5 is helping Department ID 101 in making a withdrawal of € 10 from Account ID 2
New balance in Account ID 2 is € 50080.0
pool-1-thread-5 is helping Department ID 101 in making a withdrawal of € 10 from Account ID 2
New balance in Account ID 2 is € 50070.0
New balance in Account ID 2 is € 50060.0
New balance in Account ID 2 is € 50050.0
New balance in Account ID 2 is € 50040.0
New balance in Account ID 2 is € 50030.0
New balance in Account ID 2 is € 50020.0
New balance in Account ID 2 is € 50010.0
New balance in Account ID 2 is € 50000.0
Threads have finished executing their tasks.
```

Figure(2)(b): TC02: DB before.



acc_id	acc_name	acc_balance
1		100
2		200000
3		10918
4		10000

Figure(2)(c): TC02: DB after.



acc_id	acc_name	acc_balance
1		100
2		50000
3		10918
4		10000

3. Test Case 03:

Aim:

- Demonstrate variations in load and core.
- Demonstrate **transfer funds** functionality.

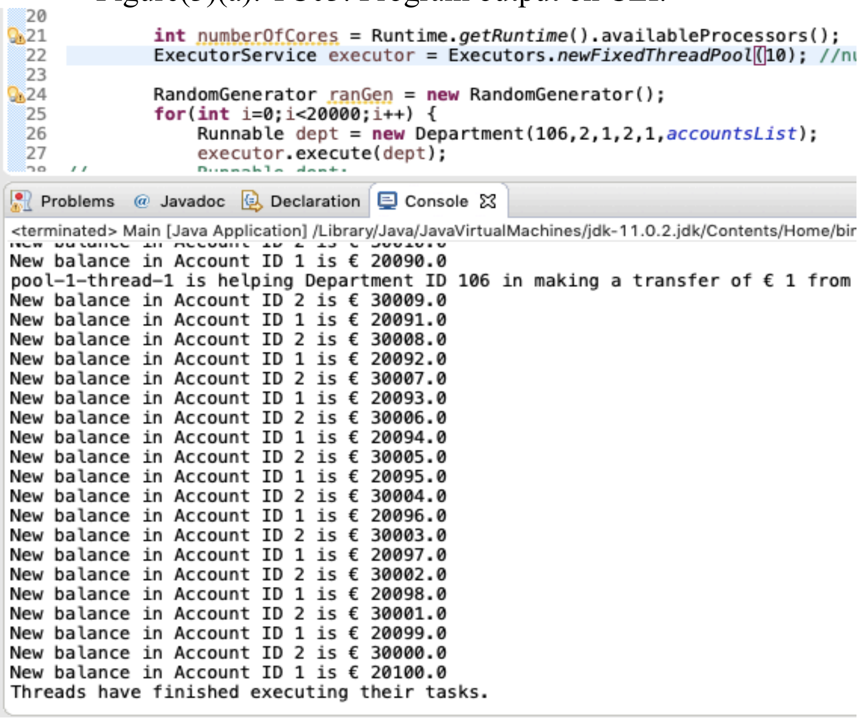
Test conditions:
Number of Cores: 10
Number of tasks (load): 20,000

Scenario: Now that the balance in Account ID 2 is €50,000; keep transfering an amount of €1 from Account ID 2 to Account ID 1. Run this task a 20,000 times.
Expected Output: Final balance in Account ID 2 should be €30,000 and in Account ID 1 should be €20,100.
Actual Output: Final balance in Account ID 2 is €30,000 and in Account ID 1 is €20,100.
Result: **Pass**
Conclusion:

- Program is **correct** even for a load of 20,000 and 10 cores.
- **Transfer funds** functionality works as expected.

Screenshot: In figure (3)(a), (3)(b), (3)(c).
NOTE: Other screenshots and output of this testcase is submitted along with the submission folder.

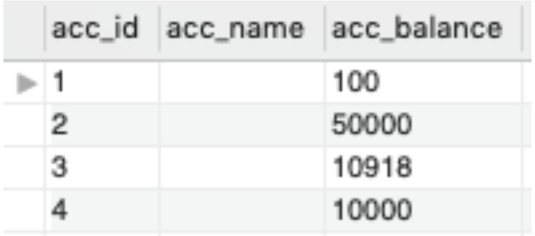
Figure(3)(a): TC03: Program output on CLI.



```
        int numberOfCores = Runtime.getRuntime().availableProcessors();
        ExecutorService executor = Executors.newFixedThreadPool(10); //nu

        RandomGenerator ranGen = new RandomGenerator();
        for(int i=0;i<20000;i++) {
            Runnable dept = new Department(106,2,1,2,1,accountsList);
            executor.execute(dept);
```

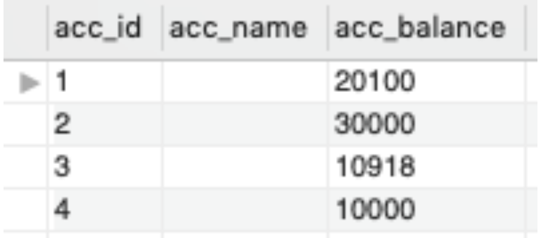
```
<terminated> Main [Java Application] /Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/bir
New balance in Account ID 1 is € 20090.0
pool-1-thread-1 is helping Department ID 106 in making a transfer of € 1 from
New balance in Account ID 2 is € 30009.0
New balance in Account ID 1 is € 20091.0
New balance in Account ID 2 is € 30008.0
New balance in Account ID 1 is € 20092.0
New balance in Account ID 2 is € 30007.0
New balance in Account ID 1 is € 20093.0
New balance in Account ID 2 is € 30006.0
New balance in Account ID 1 is € 20094.0
New balance in Account ID 2 is € 30005.0
New balance in Account ID 1 is € 20095.0
New balance in Account ID 2 is € 30004.0
New balance in Account ID 1 is € 20096.0
New balance in Account ID 2 is € 30003.0
New balance in Account ID 1 is € 20097.0
New balance in Account ID 2 is € 30002.0
New balance in Account ID 1 is € 20098.0
New balance in Account ID 2 is € 30001.0
New balance in Account ID 1 is € 20099.0
New balance in Account ID 2 is € 30000.0
New balance in Account ID 1 is € 20100.0
Threads have finished executing their tasks.
```

Figure(3)(b): TC03: DB before.



acc_id	acc_name	acc_balance
1		100
2		50000
3		10918
4		10000

Figure(3)(c): TC03: DB after.



acc_id	acc_name	acc_balance
1		20100
2		30000
3		10918
4		10000

4. Test Case 04:

Aim:

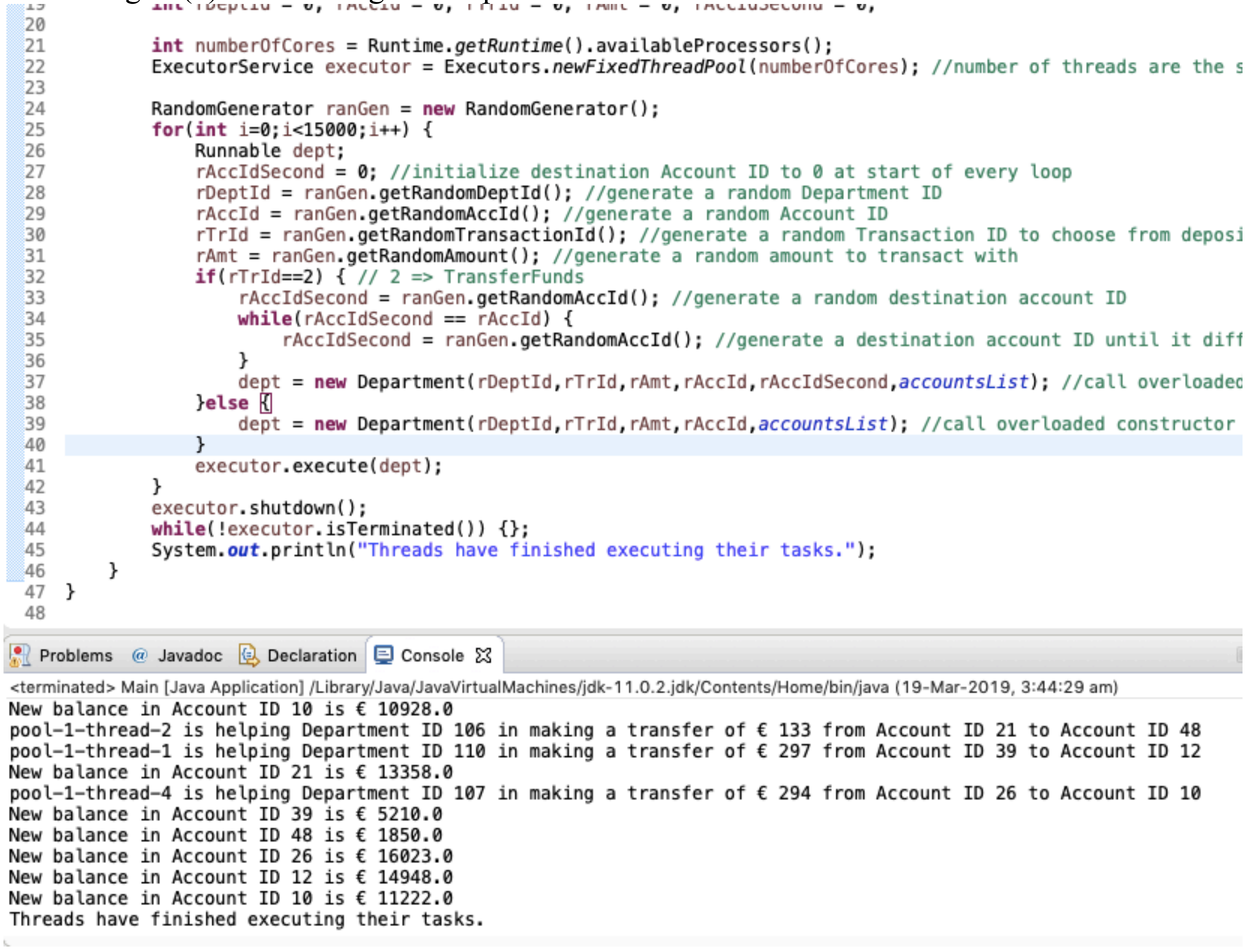
- Demonstrate the full-fledged working of the program i.e., the 10 departments making transactions (deposit, withdrawal and transfer funds) over 50 accounts. These accounts, departments, transaction types and amounts are selected at random.

Test conditions:
Number of Cores: 4
Number of tasks (load): 15,000
Scenario: 10 departments is collectively making 15,000 transactions over 50 accounts.
Expected Output: Program should run and terminate correctly.
Actual Output: Program ran and terminated correctly.
Result: **Pass**
Conclusion:

- Program is **correct** even for a **random** load of 15,000 and 4 cores.

Screenshot: In figure (4)
NOTE: Other screenshots and output of this testcase is submitted along with the submission folder.

Figure(4): TC04: Program output on CLI.



```
        int rDeptId = 0, rAccId = 0, rTrId = 0, rAmt = 0, rAccIdSecond = 0;

        int numberOfCores = Runtime.getRuntime().availableProcessors();
        ExecutorService executor = Executors.newFixedThreadPool(numberOfCores); //number of threads are the s

        RandomGenerator ranGen = new RandomGenerator();
        for(int i=0;i<15000;i++) {
            Runnable dept;
            rAccIdSecond = 0; //initialize destination Account ID to 0 at start of every loop
            rDeptId = ranGen.getRandomDeptId(); //generate a random Department ID
            rAccId = ranGen.getRandomAccId(); //generate a random Account ID
            rTrId = ranGen.getRandomTransactionId(); //generate a random Transaction ID to choose from deposi
            rAmt = ranGen.getRandomAmount(); //generate a random amount to transact with
            if(rTrId==2) { // 2 => TransferFunds
                rAccIdSecond = ranGen.getRandomAccId(); //generate a random destination account ID
                while(rAccIdSecond == rAccId) {
                    rAccIdSecond = ranGen.getRandomAccId(); //generate a destination account ID until it diff
                }
                dept = new Department(rDeptId,rTrId,rAmt,rAccId,rAccIdSecond,accountsList); //call overloadec
            }else {
                dept = new Department(rDeptId,rTrId,rAmt,rAccId,accountsList); //call overloaded constructor
            }
            executor.execute(dept);
        }
        executor.shutdown();
        while(!executor.isTerminated()) {};
        System.out.println("Threads have finished executing their tasks.");
    }
}
```

```
<terminated> Main [Java Application] /Library/Java/JavaVirtualMachines/jdk-11.0.2.jdk/Contents/Home/bin/java (19-Mar-2019, 3:44:29 am)
New balance in Account ID 10 is € 10928.0
pool-1-thread-2 is helping Department ID 106 in making a transfer of € 133 from Account ID 21 to Account ID 48
pool-1-thread-1 is helping Department ID 110 in making a transfer of € 297 from Account ID 39 to Account ID 12
New balance in Account ID 21 is € 13358.0
pool-1-thread-4 is helping Department ID 107 in making a transfer of € 294 from Account ID 26 to Account ID 10
New balance in Account ID 39 is € 5210.0
New balance in Account ID 48 is € 1850.0
New balance in Account ID 26 is € 16023.0
New balance in Account ID 12 is € 14948.0
New balance in Account ID 10 is € 11222.0
Threads have finished executing their tasks.
```

References

1. The Java Tutorials on Concurrency by Oracle.
   https://docs.oracle.com/javase/tutorial/essential/concurrency/
2. The Tutorials Point tutorials on Concurrency in JAVA
   https://www.tutorialspoint.com/java_concurrency/
3. A sample example on GitHub
   https://github.com/sabaelhilo/BankAccountExample/tree/master/src/com/assignment