Project #1, Semester 1: Developing/Testing Concurrent Systems, Thread Safety & Thread Monitoring/Management

Available Marks: 75, plus 5 bonus marks

Weighting: 50% of coursework

Submission deadline: 23:55 Sunday, 26th November, Week 10

Project Demos: Week 11

The objectives of this project are to:

- Design, implement & test a thread-safe concurrent real-world system in Java.
- Design, implement & test a basic utility that monitors and manages threads in the JVM (Java Virtual Machine).

Project overview: The project consists of two parts, which are (to a large extent) unrelated. This means that you must start working on both parts in week 3 and that you can implement both parts in parallel. Some aspects of the project involve topics that will be covered in class throughout semester 1; hence, it is very likely that your development will follow an iterative approach, i.e. implement the basic functionality first and enhance your solution later.

Part A: ***Design, implement & test a thread-safe concurrent real-world system in Java***

You are expected to cover the full development lifecycle (i.e. from requirements gathering to testing) of ***a Bank System that maintains and manages accounts for customers***. Your final product must be: **concurrent** & **thread-safe**. It is your responsibility to identify the system's requirements during the requirements gathering phase, but your end product must support:

- At least three different types of bank accounts.
- Functionality that allows account holders and bank employees to: deposit/withdraw/transfer money, check/print the current balance and account details, create/delete/edit bank accounts.

Regarding concurrency & thread-safety, your end product must be able to successfully deal with (i.e. while testing, the actual results match the expected results) the following scenarios in the context of a joint account held by two or more customers:

1. The two account holders are trying to check the balance simultaneously.
2. One account holder tries to check the balance while the other is depositing/withdrawing money.
3. The two account holders are trying simultaneously to deposit/withdraw money & then check the balance.
4. Same as 3, but at the same time a bank employee is in the process of completing a standing order in/out the account.
5. There are insufficient funds to complete a withdraw. This is an open-ended scenario and it is up to you to decide what the expected behaviour will be. Ideally (i.e. in order to achieve full marks), your system should implement a mechanism that waits for the balance to grow.
6. Two bank employees are trying simultaneously to modify the details of a bank account.

(25 Marks)

Part B: Design, implement & test a utility that monitors and manages threads in the JVM (Java Virtual Machine)

Task 1: Your task is to write & test a Java program that lists all threads in the JVM (Java Virtual Machine).

Background information: All threads in the JVM belong to a thread group, and a thread group is identified in the Java API by the `ThreadGroup` class. Thread groups are organized as a tree structure, where the root of the tree is the system thread group. The system thread group contains threads that are automatically created by the JVM, mostly for managing object references. Below the system thread group is the main thread group. The main thread group contains the initial thread in a Java program that begins execution in the `main()` method. It is also the default thread group, meaning that—unless otherwise specified—all threads you create belong to this group. It is possible to create additional thread groups and assign newly created threads to these groups. Furthermore, when creating a thread group, you may specify its parent.

Writing a program that lists all threads in the JVM will involve a careful reading of the Java API — in particular, the `java.lang.ThreadGroup` and `java.lang.Thread` classes. A strategy for

constructing this program is to first identify all thread groups in the JVM and then identify all threads within each group. To determine all thread groups, first obtain the `ThreadGroup` of the current thread, and then ascend the thread-group tree hierarchy to its root. Next, get all thread groups below the root thread group; you should find the overloaded `enumerate()` method in the `ThreadGroup` class especially helpful. Next, identify the threads belonging to each group. Again, the `enumerate()` method should prove helpful. One thing to be careful of when using the `enumerate()` method is that it expects an array as a parameter. This means that you will need to determine the size of the array before calling `enumerate()`. Additional methods in the `ThreadGroup` API should be useful for determining how to size arrays.

Your program must list each thread group and all threads within each group. When outputting each thread, your program must also list the following information (including some descriptive text):

1. The thread name
2. The thread identifier
3. The state of the thread
4. The priority of the thread
5. Whether or not the thread is a daemon

(10 Marks)

Task 2: Enhance your implementation from Part B/Task 1 so that:

- It periodically refreshes the listing of threads (including their information) and thread groups by implementing a refresh mechanism.

(5 Marks)

- It provides the following functionality:
    - Search by thread name.
    - Filter by thread group.
    - Start new thread(s).
    - Stop thread(s).

(5 Marks)

- The output appears in a tabular format using a GUI. This is an open-ended task and it is up to you to decide the "look and feel" of the interface. Your GUI should support all functionality implemented above.

(5 Marks)

Optional Task for a total of 5 bonus marks

Use your Part B implementation to monitor/manage threads while Part A is running.

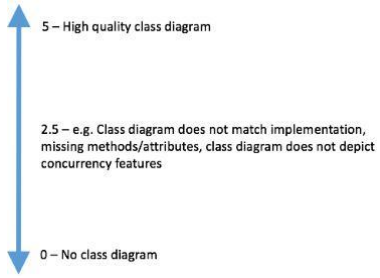
Marking Scheme

Requirements gathering & Design	[5 Marks]
Part A Implementation	[25 Marks]
Part B Implementation	[25 Marks]
Testing Strategy & Tests	[15 Marks]
Demo	[5 Marks]
Optional Task	[5 Marks]

Marking Criteria

Requirements gathering & Design [5 Marks]

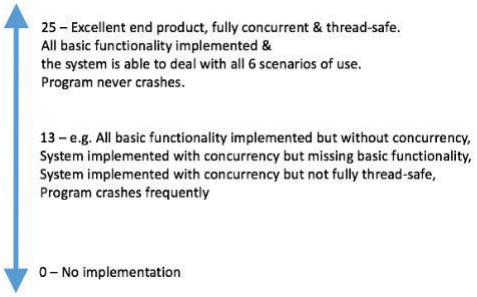
Deliverable: Class diagram (only for Part A)



Part A Implementation [25 Marks]

Deliverable: Source code of the system.

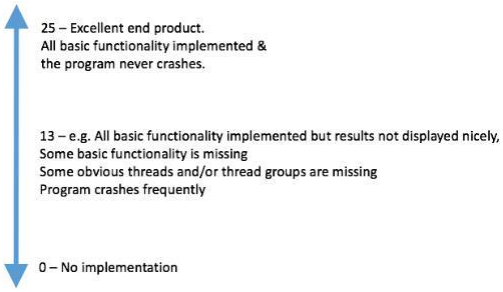
Note: Your demo will also be used as evidence to evaluate the quality of this deliverable



Part B Implementation [25 Marks]

Deliverable: Source code of the system.

Note: Your demo will also be used as evidence to evaluate the quality of this deliverable

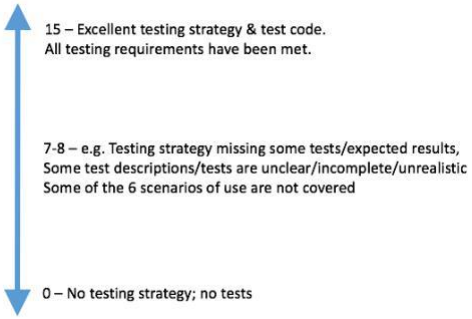


Testing Strategy & Tests [15 Marks]

Deliverables:

1. *Testing strategy document. The aim of this document is to provide a high-level description of how you plan to test your implementations (i.e. Part A & Part B). The descriptions can be in the form of pseudocode or bullet-point lists and must contain the main steps of a test and also the expected results. In order to achieve full marks, you must include:* ***at least 2 test descriptions*** *for Part A and* ***at least 1 test description*** *for Part B. For example, you can use the scenarios of use provided in Part A as a starting point, but do not just copy & paste those (doing so will not give you any marks).*

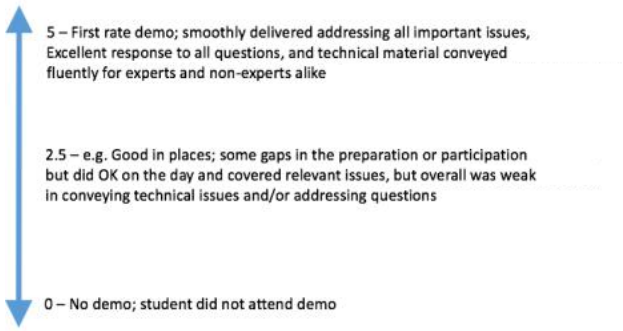
2. *Source code of your tests. These should be your actual tests. In order to achieve full marks, you must have:* ***at least 6 tests*** *for Part A that cover* ***all 6 scenarios of use*** *and* ***at least 1 test*** *for Part B. Note that these are not supposed to be JUnit tests; writing Java applications to test your implementations is just fine. Furthermore, it is OK if tests fail; this is why tests are important!*
   Note: Your demo will also be used as evidence to evaluate the quality of this deliverable



Demo [5 Marks]

Deliverable: Demos will take place in Week 11 and will involve running your systems. Furthermore, you will be asked a number of questions regarding your project.

Notes: 1. Your demo will also be used as evidence to evaluate the quality of other deliverables.



Optional Task [5 Marks]

Deliverable: By demonstration & submission of any additional source code.

Good luck!