Department of Computer Science
COS226 - Concurrent Systems

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Practical Assignment 1

- ☐ **Date Issued:** Monday, 10 August 2020
- ☐ **Date Due:** Sunday, 16 August 2019, 18:00
- ☐ **Submission Procedure:** Upload your tasks on the ClickUP website
- ☐ **Assessment:** The practical will be marked offline in the week of 17 August – 23 August 2020
- ☐ This assignment consists of 3 tasks

---

1 Introduction

1.1 Objectives and Outcomes

This assignment is meant to act as a step-by-step tutorial on the functioning of Java threads. Please read through the tutorial carefully as this will be the only material provided on the basics of multithreaded programming. From here onward, the practical assignments will assume that you know how threads work.

You must complete this assignment individually.

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 three 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 offline through a script. An opportunity for manual marking of practical submissions will be given to those who wish to query their marks. Further details on this opportunity will be communicated via the course website (ClickUP). Please note that the same marking rubric is used for auto-assessment and manual marking.

The marks for this practical are allocated as follows:

Activity	Mark
Task 1	10
Task 2	4
Task 3	6
Total	**20**

2 Implementation

Java supports multiprocessing with a number of constructs and concepts, including that of threads. In Java, the notion of a thread is encompassed by a class called `Thread`. However, there are a couple of methods by which to implement a multithreaded program, not counting the data structures that support the creation of multithreaded architectures. This assignment will introduce you to Java threads and their use.

There are two ways to create an explicit thread object in Java. The first way is to extend `java.lang.Thread`. The `TThread` class in the source code that you downloaded is an example of a thread object that extends Java's `Thread` class — carefully study the `TThread` class.

The second way is to implement the `java.lang.Runnable` interface and to pass the runnable instance to a newly constructed object of type `java.lang.Thread`. The `TRunnable` class is an example of a thread object that is created by implementing the `Runnable` interface — carefully study the `TRunnable` class.

Below are two links to give more details on the `Runnable` interface and `Thread` class.

- `Runnable`: https://docs.oracle.com/javase/7/docs/api/java/lang/Runnable.html
- `Thread`: https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html

The `ThreadDemo` class demonstrates the creation of two threads using the two different thread creation methods — carefully study this class as well.

A couple of important methods to note:

- The `Thread` class has 8 types of constructors. The parameter lists of the constructors include a variation of `ThreadGroup`, `Runnable`, and `String` objects.
- Whether you extend the `Thread` class or implement the `Runnable` interface, all thread objects should include a `run()` method that is used to perform the action for a thread. The `run()` method is executed when the `start()` method for a created thread is called.
- Threads can be forced to sleep — to temporarily cease execution — for a couple of milliseconds by using the `sleep(long)` method. This is especially useful when you want to determine the correctness of a program and want to slow down execution, or

when you want to force certain threads to back off for a couple of milliseconds before continuing execution. The `sleep(long)` method throws an `InterruptedException` which should be caught.

- `public Thread currentThread()` — returns a reference to the currently executing thread
- `public String getName()` — returns the name of the thread
- `public int getId()` — returns the identifier (ID) of the thread

Other methods and how they are used in a multithreaded program can be found in the Java API Documentation.

At the moment, in the `ThreadDemo` class, the two threads are just executing independently and are not sharing a resource. Most multithreaded programs, however, share resources and access to these shared resources should be controlled to make sure that the data is correct and not corrupted.

The `ThreadCounterDemo` class creates two threads of type `TThread` and has one shared `Counter` object. The `Counter` class has one method, namely `getAndIncrement()`. The goal of `ThreadCounterDemo` is to allow both the threads to access and increment the `Counter` object concurrently.

2.1 Task 1

For your first task you will need to change the `TThread` class to do the following:

- A thread of type `TThread` needs to be constructed with a shared `Counter` object.

  For example:

  ```
  TThread t1 = new TThread(c); // where c is the Counter object
  ```

- Instead of a simple output when the thread executes, the thread should loop **six (6)** times and with each iteration:
  - Sleep for 550 milliseconds
  - Call the shared Counter's `getAndIncrement()` method
  - Print out the default thread name, the value of the Counter that was returned by `getAndIncrement()`, and the ID of the thread. The output should conform to the following specification (where the square bracket contents are replaced with relevant data):

    [thread name] [counter value] [thread ID]

    For example: `Thread-0 1 11`
  - Make sure that the displayed output only contains the information above. The original output message of `"Thread is running..."` should not be displayed.

Note that you will need to make a couple of changes to `ThreadCounterDemo.java` to be able to test your implementation. You will, however, be required to only upload the `TThread.java` file. Therefore, make sure that your `TThread` function signatures strictly adhere to the given specifications.

Execution Notes:
When you execute `ThreadCounterDemo` a couple of times you will notice that the output is sometimes inconsistent and often incorrect. Since both threads execute the `getAndIncrement()` method 6 times, the final value of the `Counter` object should be 12. However, sometimes the final value is not 12 since threads interfered with one another and received the same value for the `Counter` object when they executed `getAndIncrement()` at the same time. This illustrates the need for Mutual Exclusion.

Upload Instructions:
Archive (zip) your modified `TThread.java` file and upload it to the **Practical 1 Task 1** submission slot on **ClickUP**.

2.2 Task 2

In Java, Mutual Exclusion can be implemented implicitly by using the `synchronized` keyword or explicitly through locks (and other safety mechanisms which we will look at later).

The `synchronized` keyword can be used to facilitate Mutual Exclusion by controlling access to a block of code or to a method.

When a method is declared a synchronized method, only one thread at a time can access the entire method. Other threads that want to access the method have to wait until the current thread leaves the method. A method is changed to a synchronized method by adding the keyword `synchronized` to the beginning of the method. Alternatively, only part of a method, or a block of code, can be synchronized by using `synchronized(this)` { } around the block.

For your second task, you should change your implementation of **Task 1** (specifically, the `Counter` class) to enforce Mutual Exclusion of the `getAndIncrement()` method using the `synchronized` keyword.

Execution Notes:
Study the output of your executed code. How does it compare to that of the previous task?

Upload Instructions:
Archive (zip) your modified `Counter.java` file and upload it to the **Practical 1 Task 2** submission slot on **ClickUP**.

2.3 Task 3

Another method for implementing Mutual Exclusion is through the use of locks. A lock is a safety mechanism that is used to control access to a block of code by only allowing one thread to acquire the lock at a specific time. While that thread holds the lock, no other thread can enter the block of code until the original thread releases the lock again.

When you write the code for your own lock, you will need to implement the `java.util.concurrent.locks.Lock` interface in order for your lock to work correctly. For this task, however, you will need to use an existing Java lock, called a `ReentrantLock (java.util.concurrent.locks.ReentrantLock)`.

A Lock has two important methods:

- `void lock()` — allows a thread to acquire the lock. If more than one thread competes for the lock at the same time, only one of them will be able to acquire the lock and the rest will have to wait until the lock is released.
- `void unlock()` — releases the lock. Only the thread that is currently holding the lock can release the lock and once it is released other threads can again try to acquire it.

To ensure that a thread releases the lock, no matter what happens during execution, the following structure should be used:

```
Lock myLock = ...

myLock.lock();
try {
    // code that should be protected by the lock
} finally {
    myLock.unlock();
}
```

When using a lock, the lock is integrated into the shared object and the locking and unlocking is done at the shared object's side, where the code that needs to be protected resides. There is only one lock that forms part of the shared object and that is also shared by the threads.

For your final task, you will need to change your implementation of **Task 2** to make use of an explicit lock instead of the `synchronized` keyword to enforce Mutual Exclusion. Add a `ReentrantLock` to the `Counter` class and use the lock's `lock()` and `unlock()` methods to protect the code that should not be executed by more than one thread at a time (i.e. the critical section).

Upload Instructions:
Archive (zip) your modified `Counter.java` file and upload it to the **Practical 1 Task 3** submission slot on **ClickUP**.