COS 226 Practical Assignment 1

- Date Issued: Monday, 27 July 2015
- Date Due: During a practical session in the week of 27 – 30 July 2015
- Submission Procedure: Show your work to a teaching assistant during one of the practical sessions
- This assignment consists of 3 tasks for a total of 3 marks.

1. Introduction

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 on forward, the practical assignments will assume that you know how threads work.

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.

2. Mark Allocation

This assignment is divided into three tasks. Each task will be marked by a teaching assistant as either correct or incorrect. There are no partial marks. Therefore:
1. your program must produce the expected output
2. your program must not throw any exceptions
3. your program must implement the correct mechanisms or algorithms, as specified by the assignment.

3. Source code

Before you begin, you should download the source code from the CS website. You are allowed to make changes to all the files in the source code.

4. Introduction

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.util.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.utilRunnable interface and to pass the runnable instance to a newly constructed object of type java.util.Thread. The TRunnable class is an
example of a thread object that is created by implementing the Runnable interface – carefully study the TRunnable class.

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:

- Whether you inherit from Thread 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 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.

Task 1

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

- Instead of a simple output when the thread executes, the thread should loop 4 times and with each iteration:
  - Sleep for 400 milliseconds
  - Call the shared Counter’s getAndIncrement() method
  - Print out the name of the thread followed by the value of the Counter that was returned by getAndIncrement()
  - For example:

    Thread-0 1  //Where Thread-0 is the name of the thread and 1 is the value of the Counter
Please note that you will also need to make a couple of changes to ThreadCounterDemo.java. Show your code and execution to a teaching assistant to get a mark and upload TThread.java to the CS website to the **Practical 1 Task 1** submission box.

**Task 2**

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 4 times, the finale value of Counter should be 8. However, sometimes the final value is not 8 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.

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 and 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 to enforce Mutual Exclusion of the `getAndIncrement()` method using the `synchronized` keyword.

Show your code and execution to a teaching assistant to get a mark and upload `Counter.java` to the CS website to the **Practical 1 Task 2** submission box.

**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 resides that need to be protected. 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() method to protect the code that you identified in task 1 that should not be executed by more than one thread at a time.

Show your code and execution to a teaching assistant to get a mark and upload Counter.java to the CS website to the Practical 1 Task 3 submission box.