Potential Problems of Sharing Data between Multiple Threads

## Thread Potential Problems

Shared resources may become corrupted if accessed simultaneously by multiple threads. When two threads access the same resource it can cause a conflict known as a race condition. An example of this, and thread interference, is where multiple threads increment a shared value, but if a thread accesses the value before it is updated by a previous thread, it will increment the wrong value, producing an incorrect answer.

The code that accesses and updates corruptible ‘shared data’ is known as a critical region. A class is thread-safe if an object cannot cause a race condition in the presence of multiple threads. Issues of deadlock and starvation must also be considered. Below are two possible solutions to making a critical region thread-safe.

## Solution One

Synchronized Methods

Synchronized methods allow only a single thread to access and execute them at a time. It is therefore ideal to have critical regions held within their own synchronized methods. Java guarantees once a thread has gained access to a synchronized method, it must finish the method before another thread can gain access to that method.

### Positives

* Easy to implement
* Ideal if it is critical that code never runs more than once at a time.
* Potential for lock elision (avoid the cost of locking on objects only visible to one thread) via escape analysis. 1
* Method is coarse, so there are less chance of deadlocks occurring. 1

### Negatives

* All synchronized methods in the same class use the same lock, reducing throughput.
* Anyone can get access to the lock, including members of other classes.
* Synchronized isn’t sufficient if there are several JVMs that need access to shared databases. 2
* Can lead to starvation as no thread sequence is set. 3

## Solution Two

java.util.concurrent

Synchronized methods allow only a single thread to access and execute them at a time. It is therefore ideal to have critical regions held within their own synchronized methods. Java guarantees once a thread has gained access to a synchronized method, it must finish the method before another thread can gain access to that method.

### Positives

* Offers greater flexible and functionality. 4
* ReentrantLock has similar functionality to Synchronised.
* Allows resource fairness, preventing deadlock. 3
* Ideal if implementation of a hand-over-hand locking protocol is required. 2

### Negatives

* More complex and takes longer to implement
* Can take longer to run.

## Comparison

With respect to performance, both mechanisms are up to the task required in this project. I originally considered the option of using the AtomicInteger method from the concurrent package. This involves using mutexs, flags raised and lowered either side of a critical region to ensure it is accessed by a single thread at a time.

However, I settled of using a synchronized method for my download protocol, as it appears to be designed for the exact need required (making the critical region thread safe) and its use is encouraged by Oracle for instances where it suits purpose. My only hesitation was based on issues of starvation, but for this project the issue wouldn’t occur with a low number of threads.

With that said, the concurrent package appears more powerful with extended functionality and I would be inclined to use it for more complex projects, but for this project it simply adds complexity where it’s not required.

## References

*1 Brian Goetz / Quiotix (2005) Escape analysis can help optimize synchronization. Available from: http://www.ibm.com/developerworks/java/library/j-jtp10185 [Accessed 02 February 2013].*

*2 David Dice (2006) ReentrantLock vs synchronized(). Available from: https://blogs.oracle.com/dave/entry/java\_util\_concurrent\_reentrantlock\_vs [Accessed 03 February 2013].*

*3 Jakob Jenkov (-) Starvation and Fairness. Available from: http://tutorials.jenkov.com/java-concurrency/starvation-and-fairness.html [Accessed 03 February 2013].*

*4 Oracle (2012) Interface Lock. Available from: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html [Accessed 07 February 2013].*