

CITS3213: CONCURRENT PROGRAMMING

Concurrency in Java II

- The basic monitor model for concurrent programming in Java presented in the last topic goes a long way.
- If you make all methods in all classes *synchronized*, and keep instance variables *private*, this is sufficient for the majority of situations involving concurrency.
- This topic treats some other aspects of concurrency in Java.
 - Synchronized static methods
 - Synchronized code blocks
 - Safe unsynchronized access to variables declared **volatile**
 - Object monitors are “reentrant”
 - Deadlocks can occur with multiple monitors
 - Other various features

Synchronized static methods

- Static methods may also be synchronized.
- There is one lock/monitor for each class.
- The lock is actually on the *class object*.

```
public class StaticCounter {
    private static int count=0;

    public static synchronized void inc() {
        count = count+1;
    }
    public static synchronized void dec() {
        count = count-1;
    }
    public static int getCount() { // not synchronized
        return count;
    }
}
```

Synchronized code blocks

- Similar to synchronized methods, but allows synchronization for any block of code, and on any object lock.
- Often used to hold locks over many method calls, or for only part of one call.
- Also used to acquire locks in order to prevent deadlock.

```
public static void main(String[] args) {
    Counter c = new Counter();
    doThis();
    synchronized (c) {
        c.inc();
        c.getCount(); // method unsynchronized
    }
    doThat();
}
```

Unsynchronized access to objects and volatile instance variables

- If the instance variables of objects are accessed by multiple threads without synchronization, the results may be odd.
- Some assignments may not be visible, even though later assignments by the same thread are.
- This is because some variables may be cached in registers.
- Also, for **long** and **double** variables, strange values may be read when *half* (32 bits) of a write has been performed.
- To avoid these issues without synchronizing (hence being forced to wait for locks), declare variables as **volatile**.

```
public class Counter {
    private volatile long count=0;

    public void reset() {
        count = 0;
    }
    public synchronized void inc() {
        count = count+1;
    }
    public int getCount() {
        return count;
    }
}
```

Reentrant monitors and deadlock

- Java monitors are “reentrant”, meaning that it is not possible for a thread to deadlock by waiting for a lock that it already holds.
- This is true even if the thread has called methods on another object, which eventually call back to the originally locked object.
- However, when threads require locks on multiple objects at the same time deadlock can occur.
- E.g. suppose we have two instances `d1`, `d2` of the following class and both `d1.eq(d2)` and `d2.eq(d1)` are called at roughly the same time.

```
class Deadlock {
    private int n;
    public Deadlock(int n) {this.n = n;}
    public synchronized boolean eq(Deadlock b) {
        return n==b.getN();    // May deadlock
    }
    public synchronized int getN() {
        return n;
    }
}
```

Deadlock prevention

- Deadlock can be prevented by requiring all threads to acquire locks in the same order.
- In practice, this is sometimes awkward.
- An alternatives is to detect deadlocks and release locks to resolve them.
- Sometimes deadlock detection is hard, and threads use timeouts (timed **waits**) instead for simplicity.
- There is no single right approach to deadlocks for all programs - often particular programs have particular solutions that work well.

Other various concurrency features of Java

- Timed waits
- Threads may be interrupted
- Can check liveness of threads
- Thread priorities (and `yield`)
- Events can be handled with “callbacks” (or “listeners”)
- Additional support in `java.util.concurrent`