Overview

- It is time to start writing some multithreaded Java programs
- We need to know how to create threads in Java and how to use the synchronisation primitives Java provides to support concurrent programming
- Java supports a rich variety of synchronisation primitives and we will only touch upon a subset
- That will finish our study of concurrency and we will then move on to look in some more detail at scheduling

Java Threads

- The Java `Thread` class implements a generic thread
- A Java thread's `run` method gives it something to do, by default it is empty and does nothing
- To have a thread do something useful, we need to override the default `run` method and substitute one of our own
- We "subclass `Thread` and override `run`" as the following example illustrates. . .

Java Threads

```
class Greeter extends Thread {
  public void run() {
    while (true) {
      System.out.println("Hello");
    }
  }
}
...
Greeter myThread = new Greeter();
myThread.start();
...
```

Java Threads

- Lab 3 introduces basic Java threads programming and there is a threads tutorial available off:

  `http://java.sun.com/docs/books/tutorial/essential`
- This tutorial will cover more than you need to know to solve your first assignment
- The tutorial covers two methods for creating threads:
  1. Subclassing `Thread` (shown above)
  2. Implementing a `Runnable` interface inside classes
- The second is the more flexible method but we will stick to the simpler subclassing `Thread` approach

Java Threads

- Every object in Java has an associated lock
- A CS is identified by the `synchronized` keyword
- Whenever a thread invokes a `synchronized` method on an object that thread locks the object (provided the lock is available)
- Other threads calling a `synchronized` method on the same object are blocked (and sleep) until the lock is freed
- The lock is automatically freed by a thread on leaving a `synchronized` method and any waiter(s) woken

Java Threads

- In the example below the producer's `run` method calls `increment` while the consumer's calls `decrement`:

```
public class Counter {
  private int value;

  public synchronized void increment() {
    value++;
  }

  public synchronized void decrement() {
    value--;
  }
}
```

Java Threads

- Lock acquisition is atomic (not much good if not!)
- Locks are reentrant so `synchronized` methods can call other `synchronized` methods on the same object and deadlock is avoided (deadlock would occur if a mutex holder requested an already held mutex)
- As usual, threads also need to be able to coordinate their efforts i.e. they will want to notify each other of certain events in order to avoid busy waiting
- They can do so with the `wait`, `notify` and `notifyAll` methods

Java Threads

- A consumer thread would call `remove`:

```
public synchronized void remove() {

  while (dataAvailable == false) {
    wait();
  }

  /* Consume an item */

  roomAvailable = true;
  notifyAll();
}
```

Java Threads

- A producer thread would call `insert`:

```
public synchronized void insert() {

  while (roomAvailable == false) {
    wait();
  }

  /* Produce an item */

  dataAvailable = true;
  notifyAll();
}
```

Java Threads

- If there is nothing available for the `remove` method to retrieve it calls `wait`, giving up the lock acquired on entering this `synchronized` method, and the calling consumer thread sleeps on a wait queue associated with the object
- Ultimately, a producer thread will make data available (note it needs the relinquished lock to do so) and call `notifyAll` waking all threads on the wait queue associated with this object (`notify` wakes only a single waiter)
- A woken thread returns from the `wait` that put it to sleep and rechecks the condition that caused it to sleep initially

Java Threads

- Before returning from `wait` woken threads compete for the object's lock (it won't be immediately available anyway since the `notifyAll` caller currently holds it)
- Eventually a consumer will return from `wait`, exit the `while` loop and consume a newly available item
- Once a consumer has removed data, producers can insert data and must be informed by a call to `notifyAll`
- All waiting producers are woken and they in turn compete for the object's lock before returning from `wait`
- Because our "buffer" has only a single slot there is constant alternate turn-taking by producer and consumer threads

wait and wait(long timeout) from J2SE API

- Causes calling thread to be suspended until another thread invokes `notify` or `notifyall` method on this object or a specified period of time has elapsed
- The calling thread must own this object's lock
- The calling thread places itself on the wait queue associated with this object and relinquishes any locks it currently holds on this object
- The thread is suspended for scheduling purposes until one of four things happen. . .

wait and wait(long timeout) from J2SE API

1. Another thread invokes `notify` for this object and this thread is chosen as the one to wake
2. Some other thread invokes `notifyall` for this object (this thread will definitely be woken)
3. Some other thread interrupts this thread
4. The specified amount of time (if any) has elapsed

- We must wrap `wait` in loop, why?
- Once marked runnable this thread must compete in the usual way to reacquire this object's lock and only then will it return from the invocation of the `wait` method

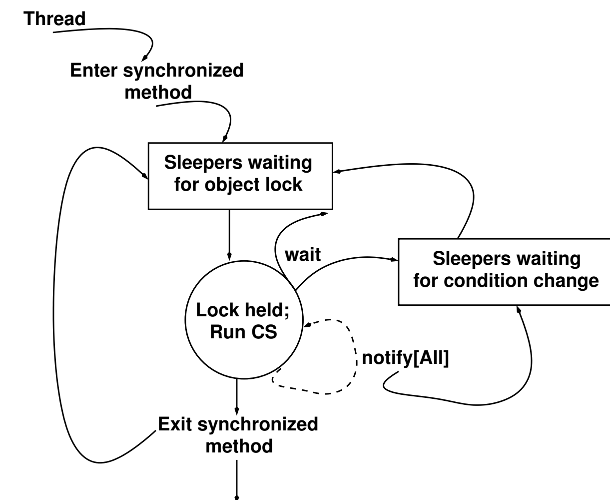
notify and notifyAll from J2SE API

- `notify` wakes up a single thread waiting on this object
- If several threads are waiting on this object, one of them is chosen to be woken - the choice is arbitrary and at the discretion of the implementation
- The woken thread will not be able to proceed until the current holder relinquishes the lock on this object and it will compete in the usual manner with any other threads that might be actively competing to synchronize on this object

notify and notifyAll from J2SE API

- The woken thread enjoys no reliable privilege or disadvantage in being the next thread to lock this object
- The method should only be called by a thread that is the owner of the the object's lock
- `notifyall` is the same as `notify` but all threads waiting on this object are woken

Java's synchronized, wait and notify



Java Semaphores

- Available as of J2SE 1.5.0
- `acquire` equates with `WAIT`
- `release` equates with `SIGNAL`
- Constructor allows the optional setting of a fairness parameter: when false the class makes no guarantee about the order in which waiting threads acquire the semaphore once released
- When set to true threads return from `acquire` in the order they arrive (FIFO) but we incur a queue management overhead