Lab 7

Signaling

Guarded Blocks with `wait()/notify()` constructs

Threads often have to coordinate their actions. The most common coordination idiom is the *guarded block*. Such a block begins by polling a condition that must be true before the block can proceed. There are a number of steps to follow in order to do this correctly.

Suppose, for example guardedJoy is a method that must not proceed until a shared variable joy has been set by another thread. Such a method could, in theory, simply loop until the condition is satisfied, but that loop is wasteful, since it executes continuously while waiting.

```
public void guardedJoy() {
    // Simple loop guard. Wastes
    // processor time. Don't do this!
    while(!joy) {}
    System.out.println("Joy has been achieved!");
}
```

A more efficient guard invokes Object.wait to suspend the current thread. The invocation of wait does not return until another thread has issued a notification that some special event may have occurred — though not necessarily the event this thread is waiting for:

```
public synchronized void guardedJoy() {
    // This guard only loops once for each special event, which may not
    // be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

Note: Always invoke wait inside a loop that tests for the condition being waited for. Don't assume that the interrupt was for the particular condition you were waiting for, or that the condition is still true.

Like many methods that suspend execution, wait can throw InterruptedException. In this example, we can just ignore that exception — we only care about the value of joy.

Why is this version of guardedJoy synchronized? Suppose d is the object we're using to invoke wait. When a thread invokes d.wait, it must own the intrinsic lock for d — otherwise an error is thrown. Invoking wait inside a synchronized method is a simple way to acquire the intrinsic lock.

When wait is invoked, the thread releases the lock and suspends execution. At some future time, another thread will acquire the same lock and invoke Object.notifyAll, informing all threads waiting on that lock that something important has happened:

```
public synchronized notifyJoy() {
    joy = true;
    notifyAll();
}
```

Some time after the second thread has released the lock, the first thread reacquires the lock and resumes by returning from the invocation of wait.

Note: There is a second notification method, notify, which wakes up a single thread. Because notify doesn't allow you to specify the thread that is woken up, it is useful only in massively parallel applications — that is, programs with a large number of threads, all doing similar chores. In such an application, you don't care which thread gets woken up.

Let's use guarded blocks to create a simple code example using the wait/notify construct.

```
class MyHouse {
    private boolean pizzaArrived = false;

    public void eatPizza(){
        synchronized(this){
            while(!pizzaArrived){
// each thread arrives here, sees that the flag has not been set
// then *releases the lock* that was obtained by synchronized(this)
// this is done so that the pizzaGuyArrived could pass through
// the synchronized(this) as well
                this.wait();
            }
        }
        System.out.println("yumyum..");
    }

    public void pizzaGuyArrived(){
        synchronized(this){
             this.pizzaArrived = true;
             this.notifyAll();
// notifyAll() wakes up *all* the threads that called this.wait()
// calling notify() would wake only *one* thread, letting the first guy
// eat the whole pizza :)
        }
    }
}
```

Some important points:

1) Avoid doing

```
if(!pizzaArrived){
    wait();
}
```

Always use `while(condition)`, because

a) threads can sporadically awake from waiting state without being notified by anyone (even when the pizza guy didn't ring the chime, somebody would decide try eating the pizza).
b) you should check for the condition again after acquiring the synchronized lock. Let's say pizza don't last forever. You awake, line-up for the pizza, but it's not enough for everybody. If you don't check, you might eat paper! :) Probably a better example would be

   ```
   while(!pizzaExists){ wait(); }.
   ```

2) You must hold the lock (synchronized) before invoking `wait/notify()`. Threads also have to acquire lock before waking.

Note: Josh Bloch, *Effective Java 2nd Edition*, Item 69: Prefer concurrency utilities to wait and notify: **Given the difficulty of using** wait **and** notify **correctly, you should use the higher-level concurrency utilities instead** [...] using wait and notify directly is like programming in "concurrency assembly language", as compared to the higher-level language provided by java.util.concurrent. **There is seldom, if ever, reason to use** wait **and** notify **in new code**.

Conditions

The Condition interface factors out the java.lang.Object monitor methods (wait(), notify(), and notifyAll()) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary Lock implementations. Where Lock replaces synchronized methods and statements, Condition replaces Object monitor methods.

Condition declares the following methods:

`void await()` – forces the current thread to wait until it's signaled or interrupted.

`boolean await(long time, TimeUnit unit)` – forces the current thread to wait until it's signaled or interrupted, or the specified waiting time elapses.

`long awaitNanos(long nanosTimeout)` – forces the current thread to wait until it's signaled or interrupted, or the specified waiting time elapses.

`void awaitUninterruptibly()` – forces the current thread to wait until it's signaled.

`boolean awaitUntil(Date deadline)` – forces the current thread to wait until it's signaled or interrupted, or the specified deadline elapses.

`void signal()` – wakes up one waiting thread.

`void signalAll()` – wakes up all waiting threads.

The previous pizza code example can be enhanced with conditions in the following way (we include only the most important snippets – you can find the full source code on ILIAS):

```
public class GourmetPizzaLoversHouse {

    ReentrantLock pizzaLock = new ReentrantLock();
    Condition meatPizzaArrivedCondition = pizzaLock.newCondition();
    Condition veganPizzaArrivedCondition = pizzaLock.newCondition();

    private boolean meatPizzaArrived = false;
    private boolean veganPizzaArrived = false;

    public void eatVeganPizza(Person person) throws InterruptedException {
        pizzaLock.lock();
        while (!veganPizzaArrived)
            veganPizzaArrivedCondition.await();
        System.out.println("vegan pizza is the best..");
        if (person.theHungriest)
            veganPizzaArrived = false; // eat the last slice
        pizzaLock.unlock();
    }

    public void eatMeatPizza(Person person) throws InterruptedException {
        pizzaLock.lock();
        while (!meatPizzaArrived)
            meatPizzaArrivedCondition.await();
        System.out.println("no way! meaty pizza is the way to go!");
        if (person.theHungriest)
            meatPizzaArrived = false; // eat the last slice
        pizzaLock.unlock();
    }

    public void pizzaGuyArrived(Pizza withPizza) {
        pizzaLock.lock();
        if (withPizza.pizzaType.equals(Pizza.PizzaType.VEGAN)) {
            veganPizzaArrived = true;
// wakes up only the vegan threads
            veganPizzaArrivedCondition.signalAll();
        } else {
            meatPizzaArrived = true;
            meatPizzaArrivedCondition.signalAll();
        }
        pizzaLock.unlock();
    }
 ...
}
```

Credits:

https://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html
http://gvsmirnov.ru/docs/presentations/java-concurrency-intro-au/#/step-33
http://www.javaworld.com/article/2078848/java-concurrency/java-concurrency-java-101-the-next-generation-java-concurrency-without-the-pain-part-2.html