Joe User learns about Java Concurrency

Joe User has two threads that want to communicate: a Producer of Great Things, and a Consumer of Great Things. Joe's first attempt looks something like this (code simplified for illustration):

Attempt 1 (naive busy-waiting)

Producer

```
GreatThing sharedGreatThing = null;
while (true) {
    sharedGreatThing = nextGreatThing();
    // Wait for consumption
    while (sharedGreatThing != null) { }
}
```

Consumer

```
while (true) {
    // Wait for sharedGreatThing
    while (sharedGreatThing == null) { }
    useGreatThing(sharedGreatThing);
    // Tell producer we have consumed it
    sharedGreatThing = null;
}
```

The program runs for a while and then stops producing output. Joe consults a psychic on stackoverflow.com who tells him that because of caching and compiler optimizations, his variable sharedGreatThing needs to be declared volatile in order for both threads to see the same value. So Joe makes the change:

Attempt 2 (busy-waiting with volatile variable)

Producer

```
volatile GreatThing sharedGreatThing = null;
while (true) {
    sharedGreatThing = nextGreatThing();
    // Wait for consumption
    while (sharedGreatThing != null) { }
}
```

Consumer

```
while (true) {
    // Wait for sharedGreatThing
    while (sharedGreatThing == null) { }
    useGreatThing(sharedGreatThing);
    // Tell producer we have consumed it
    sharedGreatThing = null;
}
```

The program runs great, but Joe notices that CPU usage is too high because of the two busy-wait loops on sharedGreatThing, so he adds a bit of a delay to the loops:

Attempt 3 (busy-waiting with delay)

Producer

```
volatile GreatThing sharedGreatThing = null;
while (true) {
    sharedGreatThing = nextGreatThing();
    // Wait for consumption
    while (sharedGreatThing != null) {
        sleep(100);
    }
}
```

Consumer

```
while (true) {
    // Wait for sharedGreatThing
    while (sharedGreatThing == null) {
        sleep(100);
    }
    useGreatThing(sharedGreatThing);
    // Tell producer we have consumed it
    sharedGreatThing = null;
}
```

Now the CPU usage is fine, but the throughput has declined because of the delays. Joe does some research and comes up with a way to avoid busy-waiting altogether:

Attempt 4 (naive notify/wait)

Producer

```
volatile GreatThing sharedGreatThing = null;
Object syncObject = new Object();
while (true) {
    sharedGreatThing = nextGreatThing();
    // Tell consumer it's ready
    syncObject.notify();
    // Wait for consumption
    syncObject.wait();
}
```

Consumer

```
while (true) {
    // Wait for sharedGreatThing
    syncObject.wait();
    useGreatThing(sharedGreatThing);
    // Tell producer we have consumed it
    syncObject.notify();
}
```

The new program works great most of the time, but sometimes it mysteriously stalls. Joe discovers that the stall is called "deadlock." He adds some debug statements to his program, and learns that the stalls happen when one thread calls notify() before the other thread calls wait(). Then Joe realizes the horrible truth about Java's most basic synchronization primitive: **Object.notify() does nothing if nobody is waiting; the notification is lost**. So Joe makes a mental note: **"Be very skeptical whenever you see Object.notify() or Object.wait() in Java code."**

Unable to give up hope on Object.notify() and Object.wait(), Joe adds back in the busy-waiting from Attempt 1:

Attempt 5 (notify/wait with busy-waiting)

Producer

```
volatile GreatThing sharedGreatThing = null;
Object syncObject = new Object();
while (true) {
    sharedGreatThing = nextGreatThing();
    // Tell consumer it's ready
    syncObject.notify();
    // Wait for consumption
    while (sharedGreatThing != null)
        syncObject.wait();
}
```

Consumer

```
while (true) {
    // Wait for sharedGreatThing
    while (sharedGreatThing == null)
        syncObject.wait();
    useGreatThing(sharedGreatThing);
    // Tell producer we have consumed it
    sharedGreatThing = null;
    syncObject.notify();
}
```

The program works, but Joe can't sleep at night knowing that his code is such an ugly hack. The next day, Joe scours the Java API looking for a way for one thread to set a condition that another thread waits for. He says to himself, "Surely this is the most common problem in concurrent programming, and Java must have some kind of Condition class that solves this problem." He soon finds the Condition interface in the Java API, but when he reads the documentation he discovers that it has the very same problem as before: **notifications are lost if nobody is waiting for them**. Joe curses Java and continues looking.

Finally he stumbles upon the oddly-named CountDownLatch class and finds a way to make it work:

Attempt 6 (CountDownLatches)

Producer

```
volatile GreatThing sharedGreatThing = null;
CountDownLatch ready = new CountDownLatch(1);
CountDownLatch consumed = new CountDownLatch(1);
while (true) {
    sharedGreatThing = nextGreatThing();
    // Tell consumer it's ready
    ready.countDown();
    // Wait for consumption
    consumed.await();
    // Reset latch
    consumed = new CountDownLatch(1);
}
```

Consumer

```
while (true) {
    // Wait for sharedGreatThing
    ready.await();
    // Reset latch
    ready = new CountDownLatch(1);
    // Do something with the Great Thing
    useGreatThing(sharedGreatThing);
    // Tell producer we have consumed it
    consumed.countDown();
}
```

Now the program works great, with high throughput and low CPU usage, but Joe is still dissatisfied with the complexity of the solution. He thinks there must be a better way. He searches the Java API again and finds ArrayBlockingQueue, which makes his next attempt a lot simpler:

Attempt 7 (ArrayBlockingQueue)

Producer

```
ArrayBlockingQueue<GreatThing> queue =
        new ArrayBlockingQueue<GreatThing>(1);
while (true) {
    queue.put(nextGreatThing());
}
```

Consumer

```
while (true) {
    useGreatThing(queue.take());
}
```

Now Joe is happy, and emails the program to his boss. He drinks a Dr. Pepper and congratulates himself on a job well-done. But then his boss storms into the room yelling, "I bought you a zillion dollar massively-parallel Plentium computer and your program runs as slow as a snail crawling through mollasses! Instead of having one producer and one consumer, you should have zillions and zillions of them, all producing and consuming at the same time!"

When his boss leaves, Joe's cubicle partner Surfer Bob says, "Dude, jump into the thread pool. Thread pools are where it's at, man. Can't go wrong with thread pools. I recommend thread pools. Thread pools are great." So Joe searches stackoverflow.com for five hours, trying to find a solution that does not involve thread pools. Finally he gives up, and submits this final attempt to his boss:

Attempt 8 (thread pool)

```
int numThreads = ZILLIONS_AND_ZILLIONS;
ExecutorService threadPool = Executors.newFixedThreadPool(numThreads);
for (int i = 0; i < numThreads; i++) {
    threadPool.execute(new Runnable() {
        public void run() {
            while (true) {
                useGreatThing(nextGreatThing());
            }
```

```
            }
        });
    }
```

Source code for this article can be found at https://github.com/Frederooni/JavaConcurrency.