Recap

- A “race condition” arises if two threads try to read and write the same data
- Might see the data in the middle of an update in a inconsistent state
  - A “race condition”: correctness depends on the update racing to completion without the reader managing to glimpse the in-progress update
  - Synchronization (also known as mutual exclusion) solves this

Java Synchronization (Locking)

```java
private Stack<String> stack = new Stack<String>();
public synchronized void doSomething() {
    // do something with s...
}
```

- Put critical operations in a `synchronized` block
- The `stack` object acts as a lock
- Only one thread can own the lock at a time

Implementing a monitor

1. Write the method names, types, and specs
   - make them synchronized
2. Write down the things you want to wait for
   - they should be predicates: you should be able to tell if they are true or false by looking only at the fields
   - document them with your class invariants and fields
3. When you need something to be true, call `wait()` while it is false
4. When you make something true, be sure to `notifyAll()` of the waiting threads.

People always misuse `wait()` and `notify()` and it leads to broken programs! Don’t! Always follow this pattern.
We illustrate these methods using an important example, which you should study and understand.

**Bounded Buffer**

Example: A baker produces breads and puts them on the shelf, like a queue. Customers take them off the shelf.

- Threads A: produce loaves of bread and put them in the queue
- Threads B: consume loaves by taking them off the queue
- This is the produce/consumer model, using a bounded buffer, the shelf (which can contain at most 20 (say) loaves of bread).

Array implementation of a queue of max size 6

Array

\[ b[0\ldots5] = [5, 3, 6, 2, 4] \]

For later purposes, we show how to implement a bounded queue — one with some maximum size — in an array.

A neat little implementation! We give you code for it on course website.

**Bounded Buffer**

```java
/** An instance maintains a bounded buffer of limited size */
class BoundedBuffer {
    ArrayQueue aq;  // bounded buffer is implemented in aq

    /** Constructor: empty bounded buffer of max size n*/
    public BoundedBuffer(int n) {
        aq = new ArrayQueue(n);
    }

    Separation of concerns:
    1. How do you implement a queue in an array?
    2. How do you implement a bounded buffer, which allows producers to add to it and consumers to take things from it, all in parallel?
```

Things to notice

- Use a while loop because we can’t predict exactly which thread will wake up “next”
- wait() waits on the same object that is used for synchronizing (in our example, `this`, which is this instance of the bounded buffer)
- Method notify() wakes up one waiting thread, notifyAll() wakes all of them up
In an ideal world...

- Bounded buffer allows producer and consumer to run concurrently, with neither blocking
  - This happens if they run at the same average rate
  - ... and if the buffer is big enough to mask any brief rate surges by either of the two

- But if one does get ahead of the other, it waits
  - This avoids the risk of producing so many items that we run out of computer memory.

About wait(), wait(n), notify(), notifyAll()

A thread that holds a lock on object OB and is executing in its synchronized code can make (at least) these calls.

1. \texttt{wait()}: It is put into set 2. Another thread from set 1 gets the lock.
2. \texttt{wait(n)}: It is put into set 2 and stays there for at least \( n \) milliseconds. Another thread from set 1 gets the lock.
3. \texttt{notify()}: Move one “possible” thread from set 2 to set 1.
4. \texttt{notifyAll()}: Move all “threads” from set 2 to set 1.

Two sets:

1. Runnable threads: Threads waiting to get the OB lock.
2. Waiting threads: Threads that called \texttt{wait} and are waiting to be notified.

Should one use notify() or notifyAll()

- Lots of discussion on this on the web! stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again
- \texttt{notify()} takes less time than \texttt{notifyAll()}
- In consumer/producer problem, if there is only one kind of consumer (or producer), probably \texttt{notify()} is OK.
- But suppose there are two kinds of bread on the shelf—and one still picks the head of the queue, if it’s the right kind of bread. Then, using \texttt{notify()} can lead to a situation in which no one can make progress. We illustrate with a proje in Eclipse, which we will put on the course website.
- \texttt{notifyAll()} always works; you need to write documentation if you optimize by using \texttt{notify()}

Using Concurrent Collections...

Java has a bunch of classes to make synchronization easier.

- It has an Atomic counter.
- It has synchronized versions of some of the Collections classes.

```
import java.util.concurrent.atomic.*;
public class Counter {
  private static AtomicInteger counter;
  public Counter() {
    counter= new AtomicInteger(0);
  }
  public static int getCount() {
    return counter.getAndIncrement();
  }
}
```

Summary

- Use of multiple processes and multiple threads within each process can exploit concurrency
  - may be real (multicore) or virtual (an illusion)
- Be careful when using threads:
  - synchronize shared memory to avoid race conditions
  - avoid deadlock
    - Even with proper locking concurrent programs can have other problems such as “livelock”
- Serious treatment of concurrency is a complex topic (covered in more detail in cs3410 and cs4410)
- Nice tutorial at http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html