1 Critical sections and atomicity

We have been seeing that sharing mutable objects between different threads is tricky. We need some kind of synchronization between the different threads to prevent them from interfering with each other in undesirable ways. For example, we saw that the following two methods on a BankAccount object got us into trouble:

```java
void withdraw(int n) {
    balance -= n;
}

void deposit(int n) {
    balance += n;
}
```

Notice that this is so even though the updates to balance are done all in one statement rather than in two as in the previous lecture. Execution of one thread may pause in the middle of that statement, so it doesn't help to write it as one statement. Two threads that are simultaneously executing withdraw and deposit, or even two threads both simultaneously executing withdraw, may cause the balance to be updated in a way that doesn't make sense.

This example shows that sometimes a piece of code needs to be executed as though nothing else in the system is making updates. Such code segments are called critical sections. They need to be executed atomically; that is, without interruption or interaction with other threads.

However, we don’t want to stop all threads just because one thread has entered a critical section. So we need a mechanism that only stops the threads that might interact with this one. The usual mechanism for achieving this is locks.

2 Mutexes and synchronized

Mutexes are mutual exclusion locks. There are two main operations on mutexes: acquire() and release(). The acquire() operation tries to acquire the mutex for the current thread. At most one thread can hold a mutex at a time. While a lock is being held by a thread, all other threads that try to acquire the lock will be blocked until the lock is released, at which point just one waiting thread will manage to acquire it.

Java supports mutexes directly. Every object has a mutex implicitly associated with it. There is no way to directly invoke the acquire() and release() operations on an object o; instead, we use the synchronized statement to acquire the object’s mutex, to perform some action, and to release the mutex:

```java
synchronized (o) {
    ...perform some action while holding o’s mutex...
}
```

The synchronized statement is useful because it makes sure that the mutex is released no matter how the statement finishing executing, even if it is through an exception. You can’t call the underlying acquire() and release() operations explicitly, but if you could, the above code using synchronized would be equivalent to this:

```java
try {
    o.acquire();
    ...perform some action while holding o’s mutex...
} finally {
    o.release();
}
```

Mutexes take up space, but a mutex is created for an object only when the object is first used for a synchronized statement, so normally they don’t add much overhead.
Using mutexes we can protect the `withdraw()` and `deposit()` methods from themselves and from each other, using the receiver object’s mutex:

```java
void withdraw(int n) {
    synchronized(this) {
        balance -= n;
    }
}

void deposit(int n) {
    synchronized(this) {
        balance += n;
    }
}
```

Because the pattern of wrapping entire method bodies in `synchronized(this)` is so common, Java has syntactic sugar for it. Declaring a method to be synchronized has the same effect:

```java
synchronized void withdraw(int n) {
    balance -= n;
}

synchronized void deposit(int n) {
    balance += n;
}
```

### 3 When is synchronization needed?

Synchronization is not free. It involves manipulating data structures, and on a machine with multiple processors (or cores), requires communication between the processors. When one is trying to make code run fast, it is tempting to cheat on synchronization. Usually this leads to disaster, as we will see when we think about when synchronization is needed.

Synchronization is needed whenever we need to rely on invariants on the state of objects, either between different fields of one or more objects, or between contents of the same field at different times. The reason for this is that without synchronization there is no guarantee that some other thread won’t be simultaneously modifying the fields in question, leading to an inconsistent view of their contents.

Synchronization is also needed when we need to make sure that one thread sees the updates caused by another thread. This is because different threads may be running on different processors. For speed, each processor has its own local copy of memory. Updates to local memory may not propagate to other processors. For example, consider two threads executing the following code in parallel:

Thread 1:

```
y = 1;
x = 1;
```

Thread 2:

```
while (x != 0) {}
print (y);
```

What possible values of y might be printed by thread 2? Naively it looks like the only possible value is 1. But without synchronization between these two threads, the update to x can be seen by thread 2 without seeing the update to y. The fact that the assignment to y happened before x does not matter. The only reliable notion of happens-before between two different threads is that enforced by explicit thread synchronization.

The short answer to when synchronization is required is that is needed for pretty much all methods of mutable objects that are shared between threads. Immutable objects that are shared between threads don’t need to be locked because no one is trying to update them.

### 4 The monitor pattern

The monitor pattern takes this observation to its natural conclusion. A monitor is an object with a built-in mutex, where all the monitor’s methods are synchronized on the mutex. This is accomplished in Java easily, because every object has a mutex, and the `synchronized` keyword enforces the monitor pattern. Java objects are essentially monitors.

The only objects that should be shared between threads are therefore immutable objects and objects protected by locks. Objects protected by locks include both monitors and objects encapsulated inside monitors, since objects encapsulated inside monitors are protected by their locks.
5 Deadlock

Monitors ensure consistency of data. But the locking they engage in can cause deadlock, a condition in which every thread is waiting to acquire a lock that some other thread is holding. For example, consider two monitors $a$ and $b$, where $a.f()$ calls $b.g()$ and vice-versa. If two threads try to call $a.f()$ and $b.g()$ respectively, the threads will acquire locks on $a$ and $b$ respectively and then deadlock trying to acquire the other lock. We can represent this situation using the diagram in Figure 1. In such a diagram, deadlocks show up as cycles in the graph.

To avoid creating cycles in the graph, the usual approach is to define an ordering on locks, and acquire locks only consistent with that ordering. For example, we might decide that $a < b$ in the lock ordering. Therefore $b$ cannot call a method of $a$ because a method of $b$ already holds a lock that is higher in the ordering.

The requirement that some locks not be held becomes a precondition of methods, which need to specify which locks may be held when the method is called. To abstract this sometimes a notion of locking level is defined. The locking level defines the highest level lock in the lock ordering that may be held when the method is called. For example

6 Locks are not enough for waiting

Locks block threads from making progress, but they are not sufficient for blocking threads in general. In general we may want to block a thread until some condition becomes true. Examples of such situations are (1) when we want to communicate information between threads (which may need to block until some information becomes available) and (2) when we want to implement our own lock abstractions.

For example, suppose we want to run two threads in parallel to compute some results and wait until both results are available. We might define a class `WorkerPair` that spawns two worker threads:
class WorkerPair extends Runnable {
    int done; // number of threads that have finished
    Object result;
    Worker() {
        done = 0;
        new Thread(this).start();
        new Thread(this).start();
    }
    public void run() {
        // note: not synchronized, to allow concurrent execution
        // do some work using synchronized methods
        synchronized(this) {
            done++;
            result = ...;
        }
    }
    Object getResult() {
        while (done < 2) {} // oops: wasteful!
        return result; // oops: not synchronized!
    }
}

We might then use this code as follows:

    w = new WorkerPair();
    Object o = w.getResult();

As the comments in the code suggest, there are two serious problems with the get\_result implementation. First, the loop on done < 2 will waste a lot of time and energy. Second, there is no synchronization ensuring that updates to result are seen.

How can we fix this? We might start by making get\_result synchronized, but this would block the final assignment to done and result in the run method. We can’t use the mutex of w to wait until done becomes 2.

7 Condition variables

The solution to the problem is to use a condition variable, which is a mechanism for blocking a thread until a condition becomes true.

Every Java object implicitly has a single condition variable tied to its mutex. It is accessed using the wait() and notify\_All() methods. (There is also a notify() method, but it should usually be avoided.)

The wait() method is used when the thread wants to wait for the condition to become true. It may only be called when the mutex is held. It atomically releases the mutex and blocks the current thread on the condition variable. The thread will only wake up and start executing when notify\_All() or notify() are called on the same condition variable. In particular, wait() will not wake up simply because the condition variable’s mutex has been released by some other thread. The other thread must call notify\_All().

Another thread should call the notify\_All() method when the condition of the condition variable might be true. Its effect is to wake up all threads waiting on the condition variable. When a thread wakes up from wait(), it immediately tries acquire the mutex. Only one thread can win; the others all block waiting for

\footnote{Java has a version of wait() that includes a timeout period after which it will automatically wake itself up. This version should usually be avoided.}
the winner to release the mutex. Eventually they acquire the mutex, though there is no guarantee that the condition is true when any of the threads awakes.

After a thread calls `wait()`, the condition it is waiting for might be true when `wait()` returns. But it need not be. Some other thread might have been scheduled first and may have made the condition false. So `wait()` is always called in a loop, like so:

```java
while (!condition) wait();
```

Failure to test the condition after `wait()` leads to what is called a wakeup–waiting race, in which threads awakened by `notifyAll()` race to observe the condition as true. The winners of the race can then spoil things for later awakeners.

Using condition variables, we can implement `getResult()` as follows:

```java
synchronized Object getResult() {
    while (done < 2) wait();
    return result;
}
```

With this implementation, the mutex is not held while the thread waits. The implementation of `run` is also modified to call `notifyAll()`:

```java
synchronized(this) {
    done++;
    result = ...
    if (done == 2) notifyAll();
}
```

The call to `notifyAll()` can be done when the mutex is held but need not be. And since the woken thread will test the condition, we need not even test it before calling `notifyAll()`:

```java
synchronized(this) {
    done++;
    result = ...
    notifyAll();
}
```

Java objects also have a `notify()` method that wakes just one thread instead of all of them. Using `notify()` is error-prone and usually should be avoided.

8 Using background threads with Swing

Swing is a single-threaded toolkit, so only one thread should try to manipulate the component hierarchy: the event dispatch thread. Any background work must be done in a separate thread, because if the event dispatch thread is busy doing work, the UI becomes unresponsive.

The `SwingWorker` class encapsulates the key functionality for starting up background threads and for obtaining results from them. This is easier than coding up your own mechanism using mutexes and condition variables.
class SwingWorker {
    /** Schedule this worker to be run on a background thread. 
     * Requires: must be called by event dispatch thread. */
    public void execute();
    /** Do some work in the worker thread. 
     * Overridable: do nothing. */
    public void doInBackground();
    /** Invoked in the event dispatch thread when doInBackground finishes. 
     * Overridable: do nothing. */
    public void done();
    ...
}

To start a background thread, an instance of a subclass of SwingWorker is created in a listener and its execute() method is called. The work it does is defined by its doInBackground method, which is overridden by the subclass. The event dispatch thread is notified of the completion of the work by a call to done(), which is overridden by the subclass.

If intermediate results need to be passed back to the event dispatch thread during execution of the background thread, there are two more methods for this purpose, called publish() and process(). The first is called in the worker thread to make results available; this causes the second method, process() to be called in the event dispatch thread, passing it whatever results were published.