Session Overview

- Wait-Notify Mechanism
- Avoiding Missed Notifications
- Waiting for the Full Timeout

Java's Wait-Notify Mechanism

- One thread efficiently signals another (or even several others).
  - One thread *waits* (using virtually zero CPU)
  - Another thread *notifies* the waiting thread

- Waiting is similar to the sleeping discussed earlier except that a lock is released (more on this later) and a notification results is a shortened "sleep" time.

- Wait-notify is supported by methods on the class `Object`:

```
public final void wait() throws InterruptedException,
                                IllegalMonitorStateException

public final void notify() throws IllegalMonitorStateException
```

- If a thread is interrupted while waiting, it will throw an `InterruptedException` (much like `Thread.sleep()` does).

- In order to call `wait()` or `notify()` on an object, the calling thread must be holding the *object*-level lock on that object.

```
Object obj = //...

obj.wait();
```
throws an `IllegalMonitorStateException` because the lock is not held.

```
obj.notify();
```
throws an `IllegalMonitorStateException` because the lock is not held.

- We rarely bother to catch `IllegalMonitorStateException` because it's a runtime exception and is only thrown when there is a bug in the code.

- The requirement that the lock is held before calling these methods ensures that no race conditions can occur. This now works:

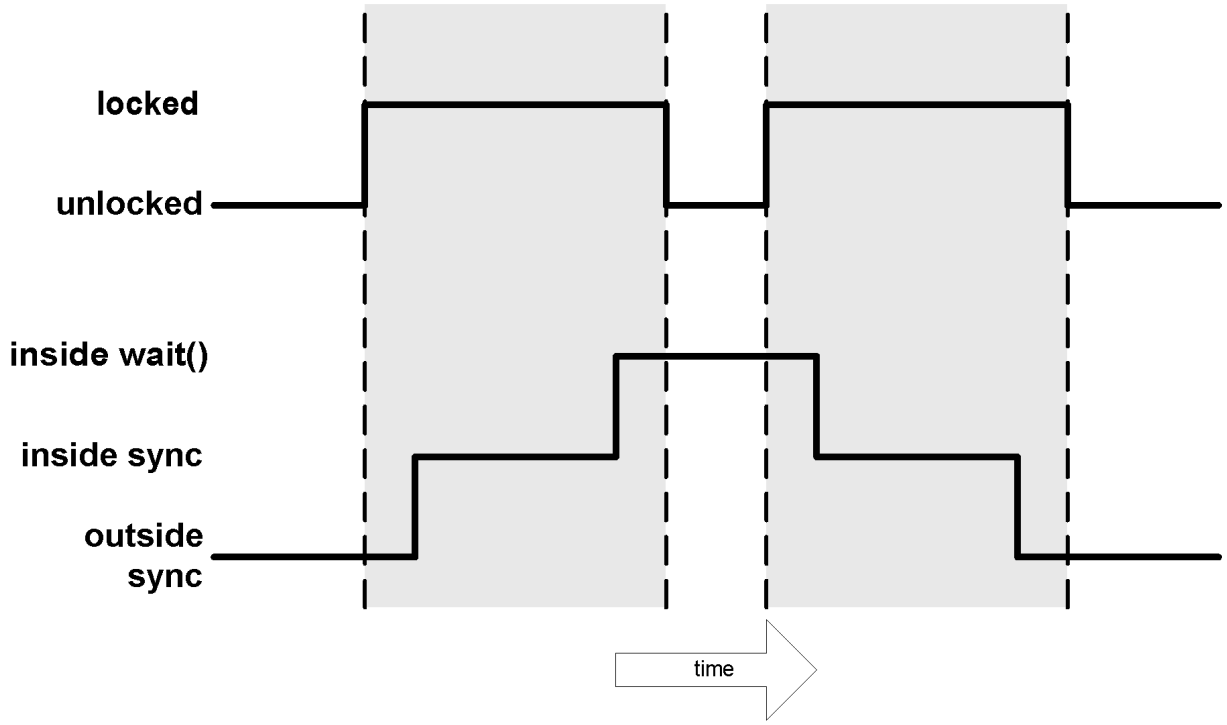
```
Object obj = //...

synchronized ( obj ) {
    obj.wait();
}

//...

synchronized ( obj ) {
    obj.notify();
}
```

- It turns out that `wait()` actually releases the object-level lock while the calling thread is inside the `wait()` method.

- This release is necessary to allow the thread calling `notify()` to be able to acquire the lock while the other waiting thread is blocked inside `wait()`.

- Calls to `notify()` do not release the lock.
- Also, unlike `wait()` calls to `Thread.sleep()` do not release the lock.

- The following timing diagram illustrates exactly when the lock is held by a thread entering a `synchronized` block of code and then calling `wait()`:



- Notice that the lock:
  - is held just before the thread is allowed inside the `synchronized` block.
  - is held slightly after `wait()` is invoked.
  - is released most of the time that the thread is inside the `wait()` method.
  - is re-acquired just before the thread returns from the `wait()` method (the thread will actually compete with other threads and block until it can regain exclusive access to the lock).
  - is held slightly after the thread leaves the `synchronized` block.

- This locking works the same way for **`synchronized`** methods:

```
public class SimpleSignal extends Object {
    public synchronized void waitForSignal()
            throws InterruptedException {

        wait();
    }

    public synchronized void signal() {
        notify();
    }
}
```

- What if more that two threads are involved? What if several threads are waiting for the signal? Imagine that three different threads have called:

```
obj.waitForSignal();
```

- It turns out that when **`notify()`** is called, only one of the waiting threads is notified (the other waiters never find out about that notification!).

- Most of the time, we'll want to notify any and all waiting threads. To do this, we use:

```
public final void notifyAll() throws IllegalMonitorStateException
```

- In 99% of cases, **`notifyAll()`** should be used to guarantee correctness. When in doubt, use **`notifyAll()`** over **`notify()`**.

```
public class SimpleSignal extends Object {
    public synchronized void waitForSignal()
            throws InterruptedException {

        wait();
    }

    public synchronized void signal() {
        notifyAll();
    }
}
```

- The only downside to using **`notifyAll()`** is that it might be wasteful (why wake up 20 threads if only one will get to proceed?). However, this is usually outweighed by the guarantee that any and all waiting threads will hear about the notification.

Missed Notification

- So far, we have assumed that one thread was already waiting when another thread came along and did the notification.
- If the thread doing the notification comes in *before* the "waiter" thread, then the "waiter" thread will have missed this notification!

- This is corrected by using a *member variable* to keep track of a state regardless of the precise timing of the `wait()` and `notify()` (or `notifyAll()`) calls.

```
public class SignalNoMiss extends Object {
    private boolean signaled = false;

    public synchronized void waitForSignal()
            throws InterruptedException {

        if ( signaled == false ) {
            wait();
        }
    }

    public synchronized void signal() {
        signaled = true;
        notifyAll();
    }
}
```

- Now, once `signal()` has been called, the `boolean` variable signaled is set and any threads that come along an invoke `waitForSignal()` return immediately—the signal is no longer missed.

Early Notification

- Sometimes a waiting thread will be notified, but the condition that the thread was waiting for has not been met.

```
public class BooleanBoxEarly extends Object {
    private boolean value;

    public BooleanBoxEarly(boolean initialValue) {
        value = initialValue;
    }

    public synchronized boolean getValue() {
        return value;
    }

    public synchronized void setValue(boolean newValue) {
        value = newValue;
        notifyAll();
    }

    public synchronized void waitUntilTrue() throws InterruptedException {
        if ( value == false ) {
            wait();
        }
    }
}
```

- This risk in this example is that imagine that we an instance:

```
BooleanBoxEarly b = new BooleanBoxEarly(false);
```

  and **threadA** executes this:

```
b.waitUntilTrue();
```

  and then the sneaky **threadB** executes this:

```
synchronized ( b ) {
    b.setValue(true);
    b.setValue(false);
}
```

  we end up with **threadA** returning thinking that the value is now **true**!

- To prevent early notification, we don't want to use **if**, we want to use **while** on the condition that we're waiting for:

```
public synchronized void waitUntilTrue() throws InterruptedException {
    while ( value == false ) {
        wait();
    }
}
```

- We can also modify **setValue()** so that notifications only occur when the value is actually changed. Although this increases efficiency, this change does not help prevent early notification (see solution of using **while** above).

```
public synchronized void setValue(boolean newValue) {
    if ( newValue != value ) {
        value = newValue;
        notifyAll();
    }
}
```

Specifying Timeouts

- In some cases, we might want to specify that the thread should only wait up to a certain amount of time for something to happen.
- For this, we can use another method on `Object`:

```
public final void wait(long msTimeout) throws InterruptedException,
                                               IllegalMonitorStateException
```

- The thread calling this method will wait until:
  - it is notified, or
  - the specified time elapses without notification, or
  - it is interrupted

- We might be tempted to try this code (but it doesn't work right!):

```
public synchronized void waitUntilTrue(
            long msTimeout
        ) throws InterruptedException {

    while ( value == false ) {
        wait(msTimeout);
    }
}
```

Trouble: what if there is an early notification again? We'll wait too long!

Waiting for the Full Timeout

- To solve this, we'll have to recalculate the *remaining* waiting time if there has been an early notification.
- Also, to indicate to the caller whether we timed out or that the value is actually now `true`, we'll return a `boolean` that follows this guideline:
  - for methods that can timeout, always return `false` to indicate that the timeout occurred. Return `true` to indicate that the condition has (finally) been met.

- This handles the problem well:

```
// return false to indicate a timeout occurred
public synchronized boolean waitUntilTrue(
            long msTimeout
        ) throws InterruptedException {

    if ( value == true ) {
        return true; // no waiting, already there
    }

    // If msTimeout is zero, then we should wait without ever timing out
    if ( msTimeout == 0L ) {
        while ( value == false ) {
            wait();
        }

        return true; // true because we did not time out
    }

    long endTime = System.currentTimeMillis() + msTimeout;
    long msRemaining = msTimeout;

    while ( ( value == false ) && ( msRemaining > 0L ) ) {
        wait(msRemaining);
        msRemaining = endTime - System.currentTimeMillis();
    }

    return ( value == true );
}
```

- Notice that the timeout value passed into `wait()`. Each wait time (`msRemaining`) is always recalculated each time a notification (or maybe an actual timeout) occurs.