The Tangible Lock
Have you a synchronized method?
The acquisition’s implicit
With the lock hiding in plain slight

Care for the lock to be tactile?
Use the Lock instead
But with responsibilities galore

A recourse when drowning in bugs?
Tread carefully with how you lock() and unlock()
and … reckon with them exceptions

Frequently asked questions from the previous class survey

- Does InterruptedException help with memory visibility like volatile?
- Are referenced objects flushed before lock acquisition?
- If the isInterrupted() passes the check just before the first statement of the loop, it completes the body?
- How many threads can acquire the static synchronized class lock?
- If you have ZERO blocking calls (with prolonged waiting durations) do you need to interrupt?
- Does the interrupt “detect” blocking calls?
Topics covered in this lecture

- Locks
- Notifications
- Wait-notify

Synchronized methods & Locks
Synchronizing methods

- **Not possible** to execute the same method in one thread while ...
  - Method is running in another thread
- If two different synchronized methods in an object are called?
  - They both require the lock of the same object
- Two or more synchronized methods of the same object *can never run in parallel* in separate threads

A lock is based on a specific instance of an object

- Not on a particular method or class
- Suppose we have 2 objects: `objectA` and `objectB` with synchronized methods `modifyData()` and `utilizeData()`
- One thread can execute `objectA.modifyData()` while another executes `objectB.utilizeData()` *in parallel*
  - Two different locks are grabbed by two different threads
  - No need for threads to wait for each other
How does a synchronized method behave in conjunction with an unsynchronized one?

- **Synchronized methods** try to grab the object lock
  - Only 1 synchronized method in an object can run at a time ... *provides data protection*

- **Unsynchronized methods**
  - Don’t grab the object lock
  - Can *execute at any time ... by any thread*
    - Regardless of whether a synchronized method is running

For a given object, at any time ...

- **Any number** of *unsynchronized methods* may be executing

- But only **1 synchronized method** can execute
Synchronizing static methods

- A lock can be obtained for each class
  - The **class lock**

- The class lock is the **object lock** of the **Class object** that models the class
  - There is only 1 **Class object per class**
  - Allows us to achieve synchronization for static methods

Object locks and class locks

- Are **not operationally related**

- The class lock can be grabbed and released **independently** of the object lock

- If a non-static synchronized method calls a static synchronized method?
  - It acquires both locks
Empty stares, from each corner of a shared prison cell
One just escapes, one’s left inside the well
And he who forgets, will be destined to remember

Nothingman, Eddie Vedder & Jeffrey Ament, Pearl Jam

**EXPLICIT LOCKING**

The synchronized keyword

- Serializes accesses to synchronized methods in an object
- Not suitable for controlling lock scope in certain situations
- Can be too primitive in some cases
Many synchronization schemes in J2SE 5.0 onwards implement the `Lock` interface

- Two important methods
  - `lock()` and `unlock()`

- Similar to using the synchronized keyword
  - Call `lock()` at the start of the method
  - Call `unlock()` at the end of the method

- Difference: we have an actual object that represents the lock
  - Store, pass around, or discard

Semantics of the using `Lock`

- If another thread owns the lock
  - Thread that attempts to acquire the lock must wait until the other thread calls `unlock()`

- Once the waiting thread acquires the lock, it returns from the `lock()` method
Using the Lock interface

```java
public class DataOperator {
    private Lock dataLock = new ReentrantLock();
    public void modifyData(byte[] newValues, int newPosition) {
        try {
            dataLock.lock();
            ... Modify values and position
        } finally {
            dataLock.unlock();
        }
    }

    public void utilizeDataAndPerformFunction() {
        try {
            dataLock.lock();
            ... Use values and position
        } finally {
            dataLock.unlock();
        }
    }
}
```

Advantages of using the Lock interface

- Grab and release locks *whenever* we want
- Now possible for *two objects to share the same lock*
  - Lock is no longer attached to the object whose method is being called
- Can be *attached to data, groups of data*, etc.
  - Not objects containing the executing methods
Advantages of explicit locking

- We can move them anywhere to **adjust lock scope**
  - Can span from a line of code to a scope that encompasses multiple methods and objects

- Lock at scope **specific to problem**
  - Not just the object

---

**Synchronized Blocks**
Much of what we accomplish with the `Lock` we can do so with the `synchronized` keyword

```java
public class DataOperator {
    public void modifyData(byte[] newValues, int newPosition) {
        synchronized(this) {
            ... Modify values and position
        }
    }

    public void utilizeDataAndPerformFunction() {
        synchronized(this) {
            ... Use values and position
        }
    }
}
```

Synchronized methods vs. Synchronized Blocks

- Possible to use only the `synchronized block` mechanism to synchronize whole method
- You decide when it's best to synchronize a block of code or the whole method
- **RULE**: Establish as small a lock scope as possible
The Lock interface [\texttt{java.util.concurrent.locks}]

```java
public interface Lock {
    public void lock();
    public void lockInterruptibly() throws InterruptedException;
    public boolean tryLock();
    public boolean tryLock(long time, TimeUnit unit) throws InterruptedException;
    public void unlock();
    public Condition newCondition();
}
```

Lock Fairness

- ReentrantLock allows locks to be granted \textit{fairly}
  - Locks are granted as close to arrival order as possible
  - Prevents lock starvation from happening

- Possibilities for granting locks
  1. First-come-first-served
  2. Allows servicing the maximum number of requests
  3. Do what's best for the platform
Objects and communications

- Every object has a lock
- Every object also includes mechanisms that allow it to be a waiting area
  - Allows communication between threads
Conditions

- One thread needs a **condition** to exist
  - Assumes another thread will **create** that condition

- When another thread creates the condition?
  - It **notifies** the first thread that has been **waiting** for that condition

---

wait(), notify() and the Object class

```java
public class Object {
    public void wait();
    public void wait(long timeout);
    public void notify();
}
```
wait(), notify() and the Object class

- Wait-and-notify mechanisms are available for every object
  - Accomplished by method invocations
- Synchronized mechanism is handled by using a keyword

Wait-and-notify relate to synchronization, but …

- It is more of a communications mechanism
- Allows one thread to communicate to another that a condition has occurred
  - Does not specify what that specific condition is
Can wait-and-notify replace the synchronized mechanism?

- No
- **Does not** solve the race condition that the synchronized mechanism solves
- Must be used in conjunction with the synchronized lock
  - Prevents race condition that exists in the wait-notify mechanism itself

A code snippet that uses wait-notify to control the execution of the thread

```java
public class Tester implements Runnable {
    private boolean done = true;

    public synchronized run() {
        while (true) {
            if (done) wait();
            else { ... Logic ... wait(100);}
        }
    }

    public synchronized void setDone(boolean b) {
        done = b;
        if (!done) notify();
    }
}
```
About the `wait()` method

- When `wait()` executes, the synchronization lock is *released*
  - By the JVM

- When a notification is received?
  - The thread needs to *reacquire* the synchronization lock before returning from `wait()`

Integration of `wait.Notify` and synchronization

- **Tightly integrated** with the synchronization lock
  - Feature not directly available to us
  - Not possible to implement this: native method

- This is typical of approach in other libraries
  - *Condition variables* for Solaris and POSIX threads require that a mutex lock be held
Details of the race condition in the wait-notify mechanism

- The first thread tests the condition and confirms that it must wait
- The second thread sets the condition
- The second thread calls notify()
  - This goes unheard because the first thread is not yet waiting
- The first thread calls wait()

How does the potential race condition get resolved?

- To call wait() or notify()
  - Obtain lock for the object on which this is being invoked
- It seems as if the lock has been held for the entire wait() invocation, but ...
  1. wait() releases lock prior to waiting
  2. Reacquires the lock just before returning from wait()
Is there a race condition during the time `wait()` releases and reacquires the lock?

- `wait()` is `tightly integrated` with the lock mechanism
- Object lock is `not freed until` the waiting thread is in a `state in which it can receive notifications`
  - System prevents race conditions from occurring here

If a thread receives a notification, is it guaranteed that condition is set?

- No
- `Prior` to calling `wait()`, `test condition` while holding lock
- `Upon returning` from `wait()` `retest` condition to see if you should `wait()` again
What if `notify()` is called and no thread is waiting?

- Wait-and-notify mechanism has no knowledge about the condition about which it notifies.
- If `notify()` is called when no other thread is waiting?
  - The notification is lost.

What happens when more than 1 thread is waiting for a notification?

- Language specification does not define which thread gets the notification.
  - Based on JVM implementation, scheduling and timing issues.
- *No way to determine* which thread will get the notification.
notifyAll()

- All threads that are waiting on an object are notified
- When threads receive this, they must work out
  1. Which thread should continue
  2. Which thread(s) should call `wait()` again
     - All threads wake up, but they **still have to reacquire the object lock**
     - Must wait for the lock to be freed

---

The contents of this slide-set are based on the following references