Overview of Java Synchronizers (Part 3)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand the capabilities provided by Java synchronizers
- Recognize Java languages features & library classes that provide
  - atomic operations & variables
  - built-in monitor objects
- Know the key synchronizers defined in the Java class library

<table>
<thead>
<tr>
<th>Java Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReentrantLock</td>
<td>A reentrant mutual exclusion lock that extends the built-in monitor lock capabilities</td>
</tr>
<tr>
<td>ReentrantReadWriteLock</td>
<td>Improves performance when resources are read much more often than written</td>
</tr>
<tr>
<td>StampedLock</td>
<td>A readers-writer lock that's more efficient than ReentrantReadWriteLock</td>
</tr>
<tr>
<td>Semaphore</td>
<td>Maintains permits that controls thread access to limited # of shared resources</td>
</tr>
<tr>
<td>ConditionObject</td>
<td>Allows Thread to block until a condition becomes true</td>
</tr>
<tr>
<td>CountDownLatch</td>
<td>Allows one or more threads to wait until a set of operations being performed in other threads complete</td>
</tr>
<tr>
<td>CyclicBarrier</td>
<td>Allows a set of threads to all wait for each other to reach a common barrier point</td>
</tr>
<tr>
<td>Phaser</td>
<td>A more flexible reusable synchronization barrier</td>
</tr>
</tbody>
</table>
Learning Objectives in this Part of the Lesson

• Understand the capabilities provided by Java synchronizers
• Recognize Java languages features & library classes that provide
  • atomic operations & variables
  • built-in monitor objects
• Know the key synchronizers defined in the Java class library
• Be aware of Java’s synchronized & concurrent collections
Overview of Java Synchronizers
Overview of Java Synchronizers

• The java.util.concurrent & java.util.concurrent.locks packages define many synchronizers
  
  • e.g., java.util.concurrent & java.util.concurrent.locks

### Overview of Java Synchronizers

<table>
<thead>
<tr>
<th>Java Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReentrantLock</td>
<td>A reentrant mutual exclusion lock that extends the built-in monitor lock capabilities</td>
</tr>
<tr>
<td>ReentrantReadWriteLock</td>
<td>Improves performance when resources are read much more often than written</td>
</tr>
<tr>
<td>StampedLock</td>
<td>A readers-writer lock that’s more efficient than ReentrantReadWriteLock</td>
</tr>
<tr>
<td>Semaphore</td>
<td>Maintains permits that control thread access to limited # of shared resources</td>
</tr>
<tr>
<td>ConditionObject</td>
<td>Allows Thread to block until a condition becomes true</td>
</tr>
<tr>
<td>CountDownLatch</td>
<td>Allows one or more threads to wait until a set of operations being performed in other threads complete</td>
</tr>
<tr>
<td>CyclicBarrier</td>
<td>Allows a set of threads to all wait for each other to reach a common barrier point</td>
</tr>
<tr>
<td>Phaser</td>
<td>A more flexible reusable synchronization barrier</td>
</tr>
</tbody>
</table>

We just cover a subset of all the Java synchronization classes
Overview of Java Synchronizers

- **ReentrantLock**
  - A mutual exclusion lock that extends built-in monitor lock capabilities

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantLock.html)
Overview of Java Synchronizers

- **ReentrantLock**
  - A mutual exclusion lock that extends built-in monitor lock capabilities
  - “Reentrant” means that the thread holding the lock can reacquire it without deadlock

See en.wikipedia.org/wiki/Reentrancy_(computing)
Overview of Java Synchronizers

• **ReentrantLock**
  - A mutual exclusion lock that extends built-in monitor lock capabilities
  - “Reentrant” means that the thread holding the lock can reacquire it without deadlock
  - Must be “fully bracketed”
    - A thread that acquires a lock must be the one to release it

See [jasleendailydiary.blogspot.com/2014/06/java-reentrant-lock.html](jasleendailydiary.blogspot.com/2014/06/java-reentrant-lock.html)
Overview of Java Synchronizers

- **ReentrantReadWriteLock**
  - Improves performance when resources read more often than written

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html)
Overview of Java Synchronizers

• **ReentrantReadWriteLock**
  - Improves performance when resources read more often than written
  - Has many features
  - Both a blessing & a curse..

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/ReentrantReadWriteLock.html)

**Reentrancy**
This lock allows both readers and writers to reacquire read or write locks in the style of a `ReentrantLock`. Non-reentrant readers are not allowed until all write locks held by the writing thread have been released.

Additionally, a writer can acquire the read lock, but not vice-versa. Among other applications, reentrancy can be useful when write locks are held during calls or callbacks to methods that perform reads under read locks. If a reader tries to acquire the write lock it will never succeed.

**Lock downgrading**
Reentrancy also allows downgrading from the write lock to a read lock, by acquiring the write lock, then the read lock and then releasing the write lock. However, upgrading from a read lock to the write lock is not possible.

**Interruption of lock acquisition**
The read lock and write lock both support interruption during lock acquisition.

**Condition support**
The write lock provides a `Condition` implementation that behaves in the same way, with respect to the write lock, as the `Condition` implementation provided by `newCondition()` does for `ReentrantLock`. This `Condition` can, of course, only be used with the write lock.

The read lock does not support a `Condition` and `readLock().newCondition()` throws `UnsupportedOperationException`.
Overview of Java Synchronizers

- **StampedLock**

- A readers-writer lock that's more efficient than a ReentrantReadWriteLock

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html)
Overview of Java Synchronizers

- **StampedLock**
  - A readers-writer lock that’s more efficient than a ReentrantReadWriteLock
  - Supports “optimistic” reads

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html)
Overview of Java Synchronizers

• **StampedLock**
  • A readers-writer lock that’s more efficient than a ReentrantReadWriteLock
  • Supports “optimistic” reads
  • Also supports “lock upgrading”

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/StampedLock.html)
Overview of Java Synchronizers

- **Semaphore**
  - Maintains permits that control thread access to limited # of shared resources

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html)
Overview of Java Synchronizers

• Semaphore
  • Maintains permits that control thread access to limited # of shared resources
  • Operations need not be fully bracketed..

```
java
Semaphore Semaphore(int)
Semaphore(int,boolean)
acquire():void
acquireUninterruptibly():void
tryAcquire():boolean
tryAcquire(long,TimeUnit):boolean
release():void
acquire(int):void
acquireUninterruptibly(int):void
tryAcquire(int):boolean
tryAcquire(int,long,TimeUnit):boolean
release(int):void
availablePermits():int
drainPermits():int
isFair():boolean
hasQueuedThreads():boolean
getQueueLength():int
toString()
```
Overview of Java Synchronizers

- **ConditionObject**
  - Allows a thread to wait until some condition become true

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html)
Overview of Java Synchronizers

• **ConditionObject**
  - Allows a thread to wait until some condition become true
  - Always used in conjunction with a ReentrantLock

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html)
Overview of Java Synchronizers

- **CountDownLatch**
  - Allows one or more threads to wait on the completion of operations in other threads

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountDownLatch.html](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CountDownLatch.html)
Overview of Java Synchronizers

• CyclicBarrier
  • Allows a set of threads to all wait for each other to reach a common barrier point

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/CyclicBarrier.html
Overview of Java Synchronizers

• Phaser

• A synchronization barrier that’s more flexible & reusable than CyclicBarrier & CountDownLatch

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Phaser.html
Overview of Java Synchronizers

- Choosing between these synchronizers involve understanding various tradeoffs between *performance* & *productivity*
Overview of Java Synchronizers

• Choosing between these synchronizers involve understanding various tradeoffs between *performance* & *productivity*

• Some synchronizers (or synchronizer methods) have more overhead
  • e.g., spin locks vs. sleep locks vs. hybrid locks

Choosing between these synchronizers involve understanding various tradeoffs between performance & productivity

• Some synchronizers (or synchronizer methods) have more overhead
• Some synchronizers are harder to program correctly than others
  • e.g., risk of deadlock from non-reentrant locking semantics

Deadlocks are problematic in object-oriented frameworks due to callbacks & complex control flows

See en.wikipedia.org/wiki/Deadlock
Overview of Java Synchronizers

- Java synchronizers differ from Java built-in monitor objects
Overview of Java Synchronizers

- Java synchronizers differ from Java built-in monitor objects, e.g.
- They are largely written in Java rather than C/C++
Overview of Java Synchronizers

• Java synchronizers differ from Java built-in monitor objects, e.g.
  • They are largely written in Java rather than C/C++
  • Some low-level methods written in native C/C++
  • e.g., compareAndSwapInt(), park(), unpark(), etc.

Concurrence

And few words about concurrency with Unsafe. compareAndSwap methods are atomic and can be used to implement high-performance lock-free data structures.

For example, consider the problem to increment value in the shared object using lot of threads.

First we define simple interface `Counter`:

```java
interface Counter {
    void increment();
    long getCounter();
}
```

Then we define worker thread `CounterClient`, that uses Counter:

```java
class CounterClient implements Runnable {
    private Counter c;
    private int num;

    public CounterClient(Counter c, int num) {
        this.c = c;
        this.num = num;
    }

    @Override
    public void run() {
        for (int i = 0; i < num; i++) {
            c.increment();
        }
    }
}
```

See mishadoff.com/blog/java-magic-part-4-sun-dot-misc-dot-unsafe
Overview of Java Synchronizers

- Java synchronizers differ from Java built-in monitor objects, e.g.
  - They are largely written in Java rather than C/C++
  - They provide *many* more features & have more powerful semantics
Overview of Java Synchronized & Concurrent Collections
Overview of Java Synchronized Collections

- By default, Java collections are not synchronized

Note that this implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally.

See docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html
Overview of Java Synchronized Collections

- By default, Java collections are not synchronized
- Thus, they are not thread-safe

Code is thread-safe if it only manipulates shared data structures in a manner that avoids race conditions by multiple concurrent threads

See en.wikipedia.org/wiki/Thread_safety
Overview of Java Synchronized Collections

- Java provides synchronized collection wrappers via static factory methods in the Collections class

<table>
<thead>
<tr>
<th>Collections Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>synchronizedCollection(coll)</td>
</tr>
<tr>
<td>synchronizedList(list)</td>
</tr>
<tr>
<td>synchronizedMap(map)</td>
</tr>
<tr>
<td>synchronizedSet(set)</td>
</tr>
</tbody>
</table>

See docs.oracle.com/javase/tutorial/collections/implementations/wrapper.html
Overview of Java Synchronized Collections

- Java provides synchronized collection wrappers via static factory methods in the Collections class, e.g.

- Ensure that method calls are thread-safe

```java
Map<Integer, String> mMap = new HashMap<>();

mMap = Collections.synchronizedMap(mMap);

Thread t1:
    mMap.put(1, "Newton");
    mMap.put(4, "Favre");
    mMap.put(7, "Elway");
    mMap.put(12, "Brady");
    mMap.put(13, "Warner");
    mMap.put(18, "Manning");

Thread t2:
    String s1 = mMap.get(12);

Thread t3:
    String s2 = mMap.get(13);

Thread t4:
    String s3 = mMap.get(18);
```
Overview of Java Synchronized Collections

- Java provides synchronized collection wrappers via static factory methods in the Collections class, e.g.
  
  ```java
  public class Collections {
      public static <K,V> Map<K,V> synchronizedMap (Map<K,V> m) {
          return new SynchronizedMap<>(m);
      }
  }
  ```

- Ensure that method calls are thread-safe
- Synchronized collections aren’t optimized for concurrent access

E.g., the Map parameter is simply wrapped by a SynchronizedMap
Overview of Java Synchronized Collections

Java provides synchronized collection wrappers via static factory methods in the Collections class, e.g.

- Ensure that method calls are thread-safe
- Synchronized collections aren’t optimized for concurrent access

**public class Collections {**

```java
public static <K,V> Map<K,V> synchronizedMap (Map<K,V> m) {
    return new SynchronizedMap<>(m);
}
```

```java
class SynchronizedMap<K,V> implements Map<K,V> ...
{
    private final Map<K,V> m; // Backing Map
    final Object mutex; // Object on which to synchronize

    SynchronizedMap(Map<K,V> m) {
        this.m = Objects.requireNonNull(m); mutex = this;
    }

    public V get(Object key) {
        synchronized (mutex) { return m.get(key); }
    }
}
```

A synchronized collection is thread-safe & governed by a single exclusion lock
Overview of Java Synchronized Collections

- Java provides synchronized collection wrappers via static factory methods in the Collections class, e.g.
  - Ensure that method calls are thread-safe
  - Synchronized collections aren’t optimized for concurrent access

```java
public class Collections {
    public static <K,V> Map<K,V> synchronizedMap (Map<K,V> m) {
        return new SynchronizedMap<>(m);
    }
}
```

```java
class SynchronizedMap<K,V> implements Map<K,V> {... {
    private final Map<K,V> m; // Backing Map
    final Object mutex; // Object on which to synchronize

    SynchronizedMap(Map<K,V> m) {
        this.m = Objects.requireNonNull(m); mutex = this;
    }

    public V get(Object key) {
        synchronized (mutex) { return m.get(key); }
    }
}
```

*Implemented by wrapping each method in a synchronized block*
Overview of Java Concurrent Collections

- Java concurrent collections provide features that are optimized for the needs of concurrent programs

These are the concurrent-aware interfaces:

- BlockingQueue
- TransferQueue
- BlockingDeque
- ConcurrentHashMap
- ConcurrentNavigableMap

Concurrent-aware classes include:

- LinkedBlockingQueue
- ArrayBlockingQueue
- PriorityBlockingQueue
- DelayQueue
- SynchronousQueue
- LinkedBlockingDeque
- LinkedTransferQueue
- CopyOnWriteArrayList
- CopyOnWriteArraySet
- ConcurrentHashMap

See docs.oracle.com/javase/tutorial/essential/concurrency/collections.html
Overview of Java Concurrent Collections

- Java concurrent collections provide features that are optimized for the needs of concurrent programs
- A concurrent collection is thread-safe, but is not governed by just a single exclusion lock

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html
Overview of Java Concurrent Collections

- Java concurrent collections provide features that are optimized for the needs of concurrent programs
- A concurrent collection is thread-safe, but is not governed by just a single exclusion lock
- They avoid *Memory Consistency Errors* by defining a “happens-before” relationship
  - e.g., between an operation that adds an object to the collection with subsequent operations that access or remove that object

See [docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html](https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html)
Overview of Java Concurrent Collections

- **BlockingQueue**
  - Define first-in-first-out (FIFO) data structures that block or time out when adding to a full queue or retrieving from an empty queue.

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/BlockingQueue.html)
Overview of Java Concurrent Collections

• ConcurrentHashMap
  • Concurrent retrievals & adjustable expected concurrent updates via OO & functional programming APIs

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ConcurrentHashMap.html
Overview of Java Concurrent Collections

- **ConcurrentHashMap**
  - Concurrent retrievals & adjustable expected concurrent updates via OO & functional programming APIs
  - Highly optimized for multi-core processors

End of Overview of Java Synchronizers (Part 3)