Overview of Java Synchronizers (Part 1)

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

<table>
<thead>
<tr>
<th>Category</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomic operations</td>
<td>An action that effectively happens all at once or not at all</td>
</tr>
<tr>
<td>Mutual exclusion</td>
<td>Allows concurrent access &amp; updates to shared data without race conditions</td>
</tr>
<tr>
<td>Coordination</td>
<td>Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.</td>
</tr>
<tr>
<td>Barrier synchronization</td>
<td>Ensures that any thread(s) must stop at a certain point &amp; cannot proceed until all other thread(s) reach this barrier</td>
</tr>
</tbody>
</table>
Overview of Java Synchronizers
Overview of Java Synchronizers

- A Java synchronizer is an object that controls the flow of cooperating threads based on its state

See en.wikipedia.org/wiki/Synchronization_(computer_science)
Overview of Java Synchronizers

- Java synchronizers ensure interactions between threads obey certain properties
Overview of Java Synchronizers

- Java synchronizers ensure interactions between threads obey certain properties, e.g.
  - Don’t corrupt shared mutable data
Overview of Java Synchronizers

- Java synchronizers ensure interactions between threads obey certain properties, e.g.
  - Don’t corrupt shared mutable data
  - Occur in the right order, at the right time, & under the right conditions
Java Synchronizers Address Inherent Complexities
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency

Inherent complexities are the “rocket science” of software development
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
    - Ensures an action happens all at once or not at all

See [en.wikipedia.org/wiki/Linearizability](en.wikipedia.org/wiki/Linearizability)
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
    - Ensures an action happens all at once or not at all
    - Operations on a field in thread_1 occur all at once wrt operations on the field in thread_2..n

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread_1</th>
<th>Thread_2</th>
<th>Long field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>initialized</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>read field</td>
<td>← 0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>increase field by 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>write back</td>
<td>→ 1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>read field</td>
<td>← 1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>increase field by 1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>write back</td>
<td>→ 2</td>
<td>2</td>
</tr>
</tbody>
</table>

*Atomicity does not occur on primitive Java data types without using synchronizers*

See [docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html](docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html)
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
    - Ensures an action happens all at once or not at all
    - Operations on a field in thread$_1$ occur all at once wrt operations on the field in thread$_2$..$_n$
    - Atomic ordering is supported by the Java atomic package

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html)
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
  - **Mutual exclusion**
    - Prevents simultaneous access to a shared resource in a critical section

See [en.wikipedia.org/wiki/Mutual_exclusion](http://en.wikipedia.org/wiki/Mutual_exclusion)
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - Mutual exclusion
    - Prevents simultaneous access to a shared resource in a critical section

Race conditions occur when a program depends on the sequence or timing of threads for it to operate properly.
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - Mutual exclusion
    - Prevents simultaneous access to a shared resource in a critical section
  - Read/write conflicts
    - If one thread reads while another thread writes concurrently, the field that's read may be inconsistent

<table>
<thead>
<tr>
<th>Time</th>
<th>Thread\textsubscript{1}</th>
<th>Thread\textsubscript{2}</th>
<th>Long field</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialized</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>read field</td>
<td></td>
<td>← 0</td>
<td></td>
</tr>
<tr>
<td>increase field by 1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>write back</td>
<td>read field</td>
<td>← 0 or 1?</td>
<td></td>
</tr>
</tbody>
</table>

Two operations *conflict* if at least one is a write
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - **Atomic ordering**
  - **Mutual exclusion**
    - Prevents simultaneous access to a shared resource in a critical section
    - Read/write conflicts
    - Write/write conflicts
      - If two threads try to write to same field concurrently, the result may be inconsistent

<table>
<thead>
<tr>
<th></th>
<th>Thread(_1)</th>
<th>Thread(_2)</th>
<th>Long field</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialized</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>read field</td>
<td>← 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read field</td>
<td></td>
<td>← 0</td>
<td></td>
</tr>
<tr>
<td>increase field by 2</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>increase field by 1</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>write back</td>
<td>write back</td>
<td>→ 1 or 2?</td>
<td></td>
</tr>
</tbody>
</table>
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - Mutual exclusion
    - Prevents simultaneous access to a shared resource in a critical section
    - Read/write conflicts
    - Write/write conflicts

See en.wikipedia.org/wiki/Memory_ordering

These problems often occur in multi-core processors with “weak” memory ordering due to core caches that allow “out-of-order” load & store operations
Java Synchronizers Address Inherent Complexities

• Java synchronizers address inherent complexities of concurrency, e.g.
  • Atomic ordering
  • Mutual exclusion
    • Prevents simultaneous access to a shared resource in a critical section
    • Read/write conflicts
    • Write/write conflicts
  • Mutual exclusion is supported by the Java locks package
    • e.g., ReentrantLock, ReentrantReadWriteLock, StampedLock, etc.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/package-summary.html
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - Mutual exclusion
  - **Coordination**
    - Ensures computations run properly
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - Mutual exclusion
  - **Coordination**
    - Ensures computations run properly, e.g.
    - In the right order

```java
% java PingPong
Ready...Set...Go!
Pong!(1)
Pong!(2)
Pong!(3)
Pong!(4)
Pong!(5)
Pong!(6)
Pong!(7)
Pong!(8)
Pong!(9)
Pong!(10)
Done!
```

See [github.com/douglasraigschmidt/LiveLessons/tree/master/PingPongApplication](https://github.com/douglasraigschmidt/LiveLessons/tree/master/PingPongApplication)
Java synchronizers address inherent complexities of concurrency, e.g.

- Atomic ordering
- Mutual exclusion

**Coordination**

- Ensures computations run properly, e.g.
  - In the right order
  - At the right time

See [en.wikipedia.org/wiki/Real-time_computing](en.wikipedia.org/wiki/Real-time_computing)
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - Mutual exclusion
  - Coordination
    - Ensures computations run properly, e.g.
      - In the right order
      - At the right time
      - Under the right conditions

See github.com/douglascraigschmidt/LiveLessons/tree/master/PalantiriManagerApplication
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - Mutual exclusion
  - Coordination
    - Ensures computations run properly
    - Coordination is supported by the Java concurrent & locks packages, as well as built-in Java monitor objects
      - e.g., ConditionObject, Semaphore, etc.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - Mutual exclusion
  - Coordination
  - **Barrier synchronization**
    - Ensures that any thread(s) must stop at a certain point & cannot proceed until all thread(s) reach the barrier

*Barrier synchronization is a variant of coordination*
Java Synchronizers Address Inherent Complexities

- Java synchronizers address inherent complexities of concurrency, e.g.
  - Atomic ordering
  - Mutual exclusion
  - Coordination
- **Barrier synchronization**
  - Ensures that any thread(s) must stop at a certain point & cannot proceed until all thread(s) reach the barrier
  - Barrier synchronization is supported by the Java concurrent package
    - e.g., CountDownLatch, CyclicBarrier, Phaser, etc.

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html
End of Overview of Java Synchronizers (Part 1)
1. Which of the following are inherent complexities of synchronization

a. Programming with limited debugging tools & environments
b. Ensuring mutual exclusion in critical sections to ensure key properties
c. Coordinating multiple threads to ensure computations run properly
d. Using low-level APIs written in C
Discussion Questions

2. Match following categories of synchronization mechanisms to their corresponding definitions

   a. Coordination  
   b. Atomic operations  
   c. Barrier synchronization  
   d. Mutual exclusion

1. An action that effectively happens all at once or not at all
2. Allows concurrent access & updates to shared data without race conditions
3. Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.
4. Ensures that any thread(s) must stop at a certain point & cannot proceed until all other thread(s) reach this barrier