Overview of Java Synchronizers

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 Lesson

- Understand the categories of 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>
Learning Objectives in this Lesson

- Understand the categories of 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 mutable 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>
**Learning Objectives in this Lesson**

- Understand the categories of 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><strong>Mutual exclusion</strong></td>
<td>Allows concurrent access &amp; updates to shared mutable 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>
Learning Objectives in this Lesson

- Understand the categories of 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 mutable 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>
**Learning Objectives in this Lesson**

- Understand the categories of 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 mutable 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
A Java synchronizer is an object used to control the flow of cooperating threads based on its state.

See [en.wikipedia.org/wiki/Synchronization_(computer_science)](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

```java
class AtomicOps {
    long mCounter = 0;

    void increment() {
        // Thread T1
        for (;;) mCounter++;
    }

    void decrement() {
        // Thread T2
        for (;;) mCounter--;
    }

    ...
}
```

*Running increment() & decrement() concurrently yields undefined behavior since mCounter is shared mutable data*
Overview of Java Synchronizers

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

```java
class AtomicOps {
    long mCounter = 0;

    synchronized void increment() {
        // Thread T1
        for (;;) mCounter++;
    }

    synchronized void decrement() {
        // Thread T2
        for (;;) mCounter--;
    }
}
```

Running increment() & decrement() concurrently yields correct behavior since mCounter is synchronized 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
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

The unsynchronized version is buggy

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

The synchronized version coordinates the threads properly

% java PlayPingPong
Ready...Set...Go!
Ping!(1)
Pong!(1)
Ping!(2)
Pong!(2)
Ping!(3)
Pong!(3)
Ping!(4)
Pong!(4)
Ping!(5)
Pong!(5)
Ping!(6)
Pong!(6)
Ping!(7)
Pong!(7)
Ping!(8)
Pong!(8)
Ping!(9)
Pong!(9)
Pong!(10)
Done!
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](http://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$_2$..$n$

### Example

<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></td>
</tr>
<tr>
<td></td>
<td>read field</td>
<td>← 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>increase field by 1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write back</td>
<td>→ 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>read field</td>
<td>← 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>increase field by 1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write back</td>
<td>→ 2</td>
<td></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 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 \(2\ldots n\)
  - 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**
    - 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\(_2\ldots n\)
    - Atomic ordering is supported by the Java atomic package
    - Atomic ordering is also supported by the Java volatile type qualifier

The volatile type qualifier ensures a variable is read from & written to main memory & not cached

See [en.wikipedia.org/wiki/Volatile_variable#In_Java](en.wikipedia.org/wiki/Volatile_variable#In_Java)
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](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

See en.wikipedia.org/wiki/Race_condition#Software
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 1</th>
<th>Thread 2</th>
<th>Long field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>initialized</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>←</td>
<td>0 or 1?</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>Time</th>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Long Field</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialized</td>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>read field</td>
<td>← 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read field</td>
<td>← 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>increase field by 2</td>
<td>0</td>
<td></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](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 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](http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/package-summary.html)
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
  - Mutual exclusion is also supported by the `synchronized` keyword in Java built-in monitor objects

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

See [github.com/douglascraigschmidt/LiveLessons/tree/master/PingPongApplication](https://github.com/douglascraigschmidt/LiveLessons/tree/master/PingPongApplication)
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

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/douglas-craig-schmidt/LiveLessons/tree/master/PalantiriManagerApplication](https://github.com/douglas-craig-schmidt/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
      - 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
    - Ensures computations run properly
    - Coordination is supported by the Java concurrent & locks packages
    - Coordination is also supported by Java built-in monitor objects

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 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](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html)
Pervasiveness of Synchronizers in Java
Pervasiveness of Java Synchronizer Classes

- Multiple layers of synchronizers are provided on the Java platform
Pervasiveness of Java Synchronizer Classes

- Multiple layers of synchronizers are provided on the Java platform, e.g.
- The Java language contains some features that synchronize threads
  - *e.g., volatile variables & built-in monitor objects*

See [en.wikipedia.org/wiki/Java](en.wikipedia.org/wiki/Java) (programming language)
Pervasiveness of Java Synchronizer Classes

- Multiple layers of synchronizers are provided on the Java platform, e.g.
  - The Java language contains some features that synchronize threads
  - Other synchronizers are provided by the Java Class Library

```
e.g., Java atomics, locks, & other synchronizers
```

See [en.wikipedia.org/wiki/Java_Class_Library](en.wikipedia.org/wiki/Java_Class_Library)
Pervasiveness of Java Synchronizer Classes

- We focus more on Java synchronization mechanisms than on Java threading mechanisms
Pervasiveness of Java Synchronizer Classes

- Synchronization complexity arises from coordinating the interactions of entities that run concurrently.
Pervasiveness of Java Synchronizer Classes

- Synchronization complexity arises from coordinating the interactions of entities that run concurrently

Java 8 parallelism frameworks may eliminate some of this complexity via “divide and conquer”
End of Overview of Java Synchronizers