Java Monitor Objects: Synchronization (Part 1)

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

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Recognize the synchronized methods/statements provided by Java build-in monitor objects to support *mutual exclusion*

Learning Objectives in this Part of the Lesson

- Mutual exclusion is used to protect shared state from corruption due to concurrent access by multiple threads
Java Synchronized Methods
Java Synchronized Methods

- The BusySynchronizedQueue class showcases Java built-in synchronization mechanisms.

```java
class BusySynchronizedQueue<E>
    implements BoundedQueue<E> {
    private LinkedList<E> mList;
    private int mCapacity;

    BusySynchronizedQueue(int capacity) {
        mList = new LinkedList<E>();
        mCapacity = capacity;
    }
    ...
}
```

See [github.com/douglascraigschmidt/POSA/tree/master/ex/M3/Queues/BusySynchronizedQueue](https://github.com/douglascraigschmidt/POSA/tree/master/ex/M3/Queues/BusySynchronizedQueue)
Java Synchronized Methods

- The BusySynchronizedQueue class showcases Java built-in synchronization mechanisms.

```java
class BusySynchronizedQueue<E> implements BoundedQueue<E> {
    private LinkedList<E> mList;
    private int mCapacity;

    BusySynchronizedQueue(int capacity) {
        mList = new LinkedList<E>();
        mCapacity = capacity;
    }

    ...  
```
The `BusySynchronizedQueue` class showcases Java built-in synchronization mechanisms. This internal state must be protected against race conditions.
The BusySynchronizedQueue class showcases Java built-in synchronization mechanisms.

```java
class BusySynchronizedQueue<E> implements BoundedQueue<E> {
    private LinkedList<E> mList;
    private int mCapacity;

    BusySynchronizedQueue(int capacity) {
        mList = new LinkedList<E>();
        mCapacity = capacity;
    }
    ...
}
```

The constructor initializes the internal state.
Java Synchronized Methods

• Methods in a built-in monitor object can be marked with the synchronized keyword:

```java
class BusySynchronizedQueue<E> implements BoundedQueue<E> {
    ...
    public synchronized boolean offer(E e)
    {
        ...
    }
    ...
    public synchronized E poll()
    {
        ...
    }
    ...
    public synchronized boolean isEmpty()
    {
        ...
    }
    ...
}
```

See [docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html](https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html)
Java Synchronized Methods

- Methods in a built-in monitor object can be marked with the `synchronized` keyword.
- A synchronized method is serialized wrt any other synchronized method in an object.

```java
class BusySynchronizedQueue<E> implements BoundedQueue<E> {
    ...
    public synchronized boolean offer(E e) {
        ... }
    public synchronized E poll() {
        ... }
    public synchronized boolean isEmpty() {
        ... }
    ...
}
```

See lesson on “Java ReentrantLock”
Java Synchronized Methods

- Methods in a built-in monitor object can be marked with the synchronized keyword.
- A synchronized method is serialized wrt any other synchronized method in an object.
- When used in the method declaration, the entire body of the method is serialized.

```java
class BusySynchronizedQueue<E>
    implements BoundedQueue<E> {

    ...

    public synchronized boolean offer(E e)
    {
        if (!isFull()) {
            mList.add(e);
            return true;
        } else
            return false;
    }

    public synchronized E poll()
    { return mList.poll(); }

    public synchronized boolean isEmpty()
    { return mList.size() == 0; }

    ...
```
The synchronized keyword is not considered to be part of a method's signature.

Synchronization is considered to be an “implementation detail”.

See gee.cs.oswego.edu/dl/cpj/mechanics.html#synchronization
Java Synchronized Methods

• The synchronized keyword is not considered to be part of a method's signature
• synchronized is *not* inherited when subclasses override superclass methods

```java
class SynchronizedQueue<E> extends BusySynchronizedQueue<E>{
    ...
    public boolean offer(E e) {
        ...
    }
    public E poll() {
        ...
    }
    public boolean isEmpty() {
        ...
    }
    ...
```

These methods will not be synchronized unless the implementation explicitly synchronizes them
Java Synchronized Methods

**Pros**
- Synchronized methods can be identified by examining the method interfaces
- The syntax is compact
  - i.e., the code is more legible
- The “method” is the unit of synchronization

**Cons**
- Synchronizes to the “intrinsic lock” (this), so it is possible for other objects to synchronize with it too
- The granularity of locking is “coarse-grained”
  - i.e., locking is a per-object/per-method basis

See [stackoverflow.com/questions/574240/is-there-an-advantage-to-use-a-synchronized-method-instead-of-a-synchronized-block/574525#574525](http://stackoverflow.com/questions/574240/is-there-an-advantage-to-use-a-synchronized-method-instead-of-a-synchronized-block/574525#574525)
End of Java Monitor Objects: Synchronization (Part 1)
Java Monitor Objects: Synchronization (Part 2)

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

• Recognize the synchronized methods/statements provided by Java build-in monitor objects to support *mutual exclusion*

```java
void m1() {
    synchronized(this) {
        ...
    }
}
```

Mutual exclusion is used to protect shared state from corruption due to concurrent access by multiple threads.
Java Synchronized Statements
Java Synchronized Statements

- Synchronized methods incur several constraints
Synchronized methods incur several constraints, e.g.

- They can yield excessive overhead due to coarse-grained serialization

**Java Synchronized Statements**

Synchronization occurs at the method level

- Thread\(\_1\) synchronizes on \(m1()\)
- Thread\(\_2\) synchronizes on \(m2()\)

**A Java Monitor Object**

- \(synchronized\) \(m1()\)
- \(synchronized\) \(m2()\)

**Wait Queue**

- \(wait()\)
- \(notify()\)
- \(notifyAll()\)

**Entrance Queue**

<<contains>>

<<contains>>
Synchronized methods incur several constraints, e.g.

- They can yield excessive overhead due to coarse-grained serialization
- Always synchronizes on the “implicit lock” (this)

May be a source of contention
Java Synchronized Statements

- e.g., consider the Java Exchanger class

```java
public class Exchanger<V> { 

    ...

    private synchronized
        void createSlot(int index){
            final Slot newSlot = new Slot();
            final Slot[] a = arena;
            if (a[index] == null)
                a[index] = newSlot;
        }

    private volatile Slot[] arena =
        new Slot[CAPACITY];
}
```

*Defines a synchronization point where threads can pair & swap elements within pairs*

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/Exchanger.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/Exchanger.html)
Java Synchronized Statements

- e.g., consider the Java Exchanger class
- One approach synchronizes at the method level

```java
public class Exchanger<V> {
    ...
    private synchronized
        void createSlot(int index){
            final Slot newSlot = new Slot();
            final Slot[] a = arena;
            if (a[index] == null)
                a[index] = newSlot;
        }

    private volatile Slot[] arena =
        new Slot[CAPACITY];
}
```
• e.g., consider the Java Exchanger class
• One approach synchronizes at the method level

```java
public class Exchanger<V> {
    ...
    private synchronized
        void createSlot(int index){
            final Slot newSlot = new Slot();
            final Slot[] a = arena;
            if (a[index] == null)
                a[index] = newSlot;
        }

    private volatile Slot[] arena =
        new Slot[CAPACITY];
}
```

Lazily create slot if this is the first time it’s accessed
Java Synchronized Statements

• e.g., consider the Java Exchanger class
  • One approach synchronizes at the method level
  • Another approach synchronizes individual statements

```java
public class Exchanger<V> {
    ...
    private
        void createSlot(int index) {
            final Slot newSlot = new Slot();
            final Slot[] a = arena;
            synchronized (this) {
                if (a[index] == null)
                    a[index] = newSlot;
            }
        }
}

private volatile Slot[] arena =
    new Slot[CAPACITY];
```

See docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html
• e.g., consider the Java Exchanger class

• One approach synchronizes at the method level

• Another approach synchronizes individual statements

```java
public class Exchanger<V> {
    ...
    private
    void createSlot(int index) {
        final Slot newSlot = new Slot();
        final Slot[] a = arena;
        synchronized (this) {
            if (a[index] == null)
                a[index] = newSlot;
        }
    }
}
```

private volatile Slot[] arena =
    new Slot[CAPACITY];

Synchronized statements are “finer-grained” than synchronized methods
Java Synchronized Statements

- e.g., consider the Java Exchanger class
- One approach synchronizes at the method level
- Another approach synchronizes individual statements

```java
public class Exchanger<V> {
    ...
    private
        void createSlot(int index){
            final Slot newSlot = new Slot();
            final Slot[] a = arena;
            synchronized (this) {
                if (a[index] == null)
                    a[index] = newSlot;
            }
        }
}

private volatile Slot[] arena =
    new Slot[CAPACITY];
```
Java Synchronized Statements

• e.g., consider the Java Exchanger class
  • One approach synchronizes at the method level
  • Another approach synchronizes individual statements
  • “Intrinsic lock” is often used to synchronize a statement

```java
public class Exchanger<V> {
    ...
    private
        void createSlot(int index){
            final Slot newSlot = new Slot();
            final Slot[] a = arena;
            synchronized (this) {
                if (a[index] == null)
                    a[index] = newSlot;
            }
        }
    
    private volatile Slot[] arena =
        new Slot[CAPACITY];
}
```

Only this statement is serialized via the “intrinsic lock”
Java Synchronized Statements

- e.g., consider the Java Exchanger class
  - One approach synchronizes at the method level
  - Another approach synchronizes individual statements
    - “Intrinsic lock” is often used to synchronize a statement
    - “Explicit lock” synchronization can also be used

```java
public class Exchanger<V> {
    ...
    private
        void createSlot(int index)
        {
            final Slot newSlot = new Slot();
            final Slot[] a = arena;
            synchronized (a) {
                if (a[index] == null)
                    a[index] = newSlot;
            }
        }

    private volatile Slot[] arena =
        new Slot[CAPACITY];
}
```

Can also synchronize using an explicit object
Java Synchronized Statements

- e.g., consider the Java Exchanger class
  - One approach synchronizes at the method level
  - Another approach synchronizes individual statements
    - “Intrinsic lock” is often used to synchronize a statement
    - “Explicit lock” synchronization can also be used
      - e.g., in situations where the intrinsic lock is too limited or too contended

```java
public class Exchanger<V> {
    ...
    private
        void createSlot(int index){
            final Slot newSlot = new Slot();
            final Slot[] a = arena;
            synchronized (a) {
                if (a[index] == null)
                    a[index] = newSlot;
            }
        }
    private volatile Slot[] arena = new Slot[CAPACITY];
}
```

Can also synchronize using an explicit object

See www.dre.vanderbilt.edu/~schmidt/PDF/specific-notification.pdf
Java Synchronized Statements

• More info is available in the online Java documentation

Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the intrinsic lock or monitor lock. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to acquire the object's intrinsic lock before accessing them, and then release the intrinsic lock when it's done with them. A thread is said to own the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock.

Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the class object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

Synchronized Statements

Another way to create synchronized code is with synchronized statements. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```java
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    namelist.add(name);
}
```

See [docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html](https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html)
Java Synchronized Statements

**Pros**
- Allows a private field to be used as the synchronizer
  - i.e., “hides” implementation details
- Synchronized statements can be found by searching references to a variable via the IDE

**Cons**
- The syntax is more complicated
  - i.e., code is harder to read

Implementing the Double-Checked Locking Pattern
Synchronized statements can be used to implement patterns like *Double-Checked Locking*

```java
public class Exchanger<V> {
    ...
    private void createSlot(int index) {
        final Slot newSlot = new Slot();
        final Slot[] a = arena;
        synchronized (a) {
            if (a[index] == null)
                a[index] = newSlot;
        }
    }

    private Object doExchange(...) {
        ...
        final Slot slot = arena[index];
        if (slot == null)
            // Lazily initialize slots
            createSlot(index);

        private volatile Slot[] arena =
                new Slot[CAPACITY];
```

• Synchronized statements can be used to implement patterns like **Double-Checked Locking**

• Synchronization is done "lazily" when initialization is first performed

```java
public class Exchanger<V> {
    ...
    private void createSlot(int index){
        final Slot newSlot = new Slot();
        final Slot[] a = arena;
        synchronized (a) {
            if (a[index] == null)
                a[index] = newSlot;
        }
    }

    private Object doExchange(...) {
        ...
        final Slot slot = arena[index];
        if (slot == null)
            // Lazily initialize slots
            createSlot(index);

        private volatile Slot[] arena =
            new Slot[CAPACITY];
    }
```
Synchronized statements can be used to implement patterns like **Double-Checked Locking**.

Synchronization is done “lazily” when initialization is first performed.

```java
public class Exchanger<V> {
    ...
    private void createSlot(int index){
        final Slot newSlot = new Slot();
        final Slot[] a = arena;
        synchronized (a) {
            if (a[index] == null)
                a[index] = newSlot;
        }
    }

    private Object doExchange(...)
    {
        final Slot slot = arena[index];
        if (slot == null)
            // Lazily initialize slots
            createSlot(index);

        private volatile Slot[] arena =
            new Slot[CAPACITY];
    }
```
public class Exchanger<V> { 
...
private void createSlot(int index) { 
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (a) {
        if (a[index] == null)
            a[index] = newSlot;
    }
}

private Object doExchange(...) { 
...
    final Slot slot = arena[index];
    if (slot == null)
        // Lazily initialize slots
        createSlot(index);
    // ...
}

private volatile Slot[] arena =
    new Slot[CAPACITY];

• Synchronized statements can be used to implement patterns like Double-Checked Locking
• Synchronization is done "lazily" when initialization is first performed

Only create a slot if the current slot is null
• Synchronized statements can be used to implement patterns like Double-Checked Locking

• Synchronization is done "lazily" when initialization is first performed

```java
public class Exchanger<V> {
    ...
    private void createSlot(int index) {
        final Slot newSlot = new Slot();
        final Slot[] a = arena;
        synchronized (a) {
            if (a[index] == null)
                a[index] = newSlot;
        }
    }

    private Object doExchange(...) {
        ...
        final Slot slot = arena[index];
        if (slot == null)
            // Lazily initialize slots
            createSlot(index);

        private volatile Slot[] arena =
            new Slot[CAPACITY];
    }
```
End of Java Monitor
Objects: Synchronization
(Part 2)