Java Monitor Object
Synchronized Statements

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 how the synchronized methods/statements provided by Java built-in monitor objects support *mutual exclusion*

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

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

See previous lessons on "Java Synchronized Methods"
Synchronized methods incur several constraints, e.g.
- They can yield excessive overhead due to coarse-grained serialization

*Synchronization occurs at the method level*
Java Synchronized Statements

- Synchronized methods incur several constraints, e.g.
  - They can yield excessive overhead due to coarse-grained serialization
  - Always synchronizes on the one & only “implicit lock” (i.e., this)

May be a source of contention
• 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 src/share/classes/java/util/concurrent/Exchanger.java
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];
}
```

*Synchronized methods are “course-grained”*
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](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];
```

Create slot outside of lock to narrow the synchronization region
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

```
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
```

See stackoverflow.com/questions/3369287/what-is-the-difference-between-synchronized-on-lockobject-and-using-this-as-the
• 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., when the intrinsic lock is too limited or too contended

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

• Pros of synchronized statements

See stackoverflow.com/questions/574240/is-there-an-advantage-to-use-a-synchronized-method-instead-of-a-synchronized-blo/574525#574525
Pros & Cons of Java Synchronized Statements

- **Pros of synchronized statements**
  - Allows a private field to be used as the synchronizer

Exchanger<Long> e = new Exchanger<>();

// Thread T1
for (;;) {
  ... e.exchange(v);
}

// Thread T2
synchronized(e) {
  ... 
}

Will not keep Thread T1 from accessing e's critical section
Cons of synchronized statements
Pros & Cons of Java Synchronized Statements

- **Cons of synchronized statements**
  - The syntax is a bit more complicated

```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;
            }
        }
    }
    ...
}

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;
        }
    }
    ...
}
```

*This code is harder to understand*
Implementing the Double-Checked Locking Pattern
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];
}
```

See [en.wikipedia.org/wiki/Lazy_initialization](http://en.wikipedia.org/wiki/Lazy_initialization)
Implementing the Double-Checked Locking Pattern

- 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];
}
```

Double-Checked Locking optimization is done here
Implementing the Double-Checked Locking Pattern

- 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];
    }
}
```

There’s no need to synchronize this check since reference reads & writes are atomic

See [docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.7](docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.7)
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];
}
End of Java Monitor Object
Synchronized Statements