The Java Monitor Object
Synchronized Method Example

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*

• Understand how to fix race conditions in the buggy concurrent Java app by using synchronized methods

The use of synchronized methods only provides a partial solution, however...
Partial Solution Using Java Synchronized Methods
Partial Solution Using Java Synchronized Methods

See en.wikipedia.org/wiki/Crazy_Horse_Memorial
A concurrent producer/consumer app that passes messages via the class “BusySynchronizedQueue”

See github.com/douglascraigschmidt/POSA/tree/master/ex/M3/Queues/BusySynchronizedQueue
Partial Solution Using Java Synchronized Methods

- The BusySynchronizedQueue is modeled on the Java ArrayBoundedQueue

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBoundedQueue.html
Partial Solution Using Java Synchronized Methods

- UML class diagram showing the design of the BusySynchronizedQueue

See [gM3/Queues/BusySynchronizedQueue/app/src/main/java/edu/vandy/busysynchronizedqueue/model](gM3/Queues/BusySynchronizedQueue/app/src/main/java/edu/vandy/busysynchronizedqueue/model)
Partial Solution Using Java Synchronized Methods

- UML sequence diagram of the BusySynchronizedQueue unit test

Partial Solution Using Java Synchronized Methods

- UML sequence diagram of the BusySynchronizedQueue unit test

The main thread coordinates the other threads in the test.
Partial Solution Using Java Synchronized Methods

- UML sequence diagram of the BusySynchronizedQueue unit test

The consumer & producer threads generate & process messages sent via the BusySynchronizedQueue, respectively
Partial Solution Using Java Synchronized Methods

- UML sequence diagram of the BusySynchronizedQueue unit test

Although the test runs correctly (since offer() & poll() are synchronized) it is inefficient due to the “busy waiting”!!

Although the test runs correctly (since offer() & poll() are synchronized) it is inefficient due to the “busy waiting”!!
Implementation of the BusySynchronizedQueue
Implementation of the BusySynchronizedQueue

- Java synchronized methods protects critical sections from concurrent access

```java
class BusySynchronizedQueue<E> implements SimpleBlockingQueue<E> {
    private List<E> mList;
    private int mCapacity;

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

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

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

See [github.com/douglascraigschmidt/POSA/tree/master/ex/M3/Queues/BusySynchronizedQueue](https://github.com/douglascraigschmidt/POSA/tree/master/ex/M3/Queues/BusySynchronizedQueue)
Implementation of the BusySynchronizedQueue

- Java synchronized methods protects critical sections from concurrent access

```java
class BusySynchronizedQueue<E>
    implements SimpleBlockingQueue<E> {
    private List<E> mList;
    private int mCapacity;

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

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

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

    ...
```

A constructor is only called once in one thread so there won’t be race conditions
Implementation of the BusySynchronizedQueue

- Java synchronized methods protects critical sections from concurrent access.

```java
class BusySynchronizedQueue<E> {
    implements SimpleBlockingQueue<E> {
        private List<E> mList;
        private int mCapacity;

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

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

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

        ...
    }
}
```

Only one synchronized method at a time can be active in any given object.
Implementation of the BusySynchronizedQueue

- Java synchronized methods protects critical sections from concurrent access

```java
class BusySynchronizedQueue<E> implements SimpleBlockingQueue<E> {
    private List<E> mList;
    private int mCapacity;

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

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

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

May be a liability for concurrently accessed objects, e.g., double-ended queues implemented as linked lists

See cs.nyu.edu/courses/fall07/G22.2631-001/lists.slides2.pdf
Implementation of the BusySynchronizedQueue

- Adding the synchronized keyword has two effects

```java
class BusySynchronizedQueue<E> implements SimpleBlockingQueue<E> {
    private List<E> mList;
    private int mCapacity;

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

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

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

See [docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html](https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html)
Implementation of the BusySynchronizedQueue

- Adding the synchronized keyword has two effects

```java
class BusySynchronizedQueue<E> {
    implements SimpleBlockingQueue<E> {
    private List<E> mList;
    private int mCapacity;

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

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

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

*Invocations of offer() & poll() on the same object can’t interleave*

i.e., each synchronized method is “atomic”
• Adding the synchronized keyword has two effects

```java
class BusySynchronizedQueue<E> implements SimpleBlockingQueue<E> {
    private List<E> mList;
    private int mCapacity;

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

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

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

Establishes a “happens-before” relation to ensure visibility of state changes to all threads

See [en.wikipedia.org/wiki/Happened-before](en.wikipedia.org/wiki/Happened-before)
End of the Java Monitor
Object Synchronized
Method Example