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
Learning Objectives in this Part of the Lesson

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

Mutual exclusion is used to protect shared state from corruption due to concurrent access by multiple threads.
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/douglasraigschmidt/POSA/tree/master/ex/M3/Queues/BusySynchronizedQueue](https://github.com/douglasraigschmidt/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;
    }
    ...
```

<<Java Interface>>

```
interface BoundedQueue<E> {
    void put(E);  // put(int)
    E take();    // take()
    E poll();   // poll()
    boolean offer(E);  // offer()
    boolean isEmpty();  // isEmpty()
    boolean isFull();   // isFull()
    int size();    // size()
}
```
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;
    }
    ...
}
```

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>()
        mList = new LinkedList<E>()
    }
}
```

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](http://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 Synchronized Methods

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

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

*Synchronization is considered to be an “implementation detail”*

See [gee.cs.oswego.edu/dl/cpj/mechanics.html#synchronization](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 of synchronized methods**

- 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

See [stackoverflow.com/questions/574240/is-there-an-advantage-to-use-a-synchronized-method-instead-of-a-synchronized-blo/574525#574525](stackoverflow.com/questions/574240/is-there-an-advantage-to-use-a-synchronized-method-instead-of-a-synchronized-blo/574525#574525)
Java Synchronized Methods

- **Cons of synchronized methods**
  - 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-blo/574525#574525](stackoverflow.com/questions/574240/is-there-an-advantage-to-use-a-synchronized-method-instead-of-a-synchronized-blo/574525#574525)
End of Java Monitor
Objects: Synchronization
(Part 1)
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
Synchronized methods incur several constraints.
Java Synchronized Statements

- Synchronized methods incur several constraints, e.g.
  - They can yield excessive overhead due to coarse-grained serialization

Synchronization occurs at the method level
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 grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/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];
```
• 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];
}
```
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
```
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];
```

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”
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];
```
Java Synchronized Statements

• e.g., consider the Java Exchanger class
  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;
      }
    }
  }

• 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

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

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

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](docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html)
**Java Synchronized Statements**

- **Pros of synchronized statements**
  - 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

Java Synchronized Statements

- **Cons of synchronized statements**
  - 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];
    }
}
```

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];
```
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];
    }
```
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];
    }
}
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];
}
```
End of Java Monitor
Objects: Synchronization
(Part 2)
Java Monitor Objects: Synchronization (Part 3)

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 the 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
Partial Solution Using Java Synchronized Methods

- 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](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

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

The offer() & poll() methods are synchronized, so the test runs correctly, but is inefficient due to “busy waiting”!!
Implementation of the BusySynchronizedQueue
Implementation of the BusySynchronizedQueue

• Java synchronized methods protects critical sections from concurrent access

```java
class BusySynchronizedQueue<E>
    implements BoundedQueue<E> {
    private ListedList<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(); }
    ...
```

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 protect critical sections from concurrent access.

**Implementation of the BusySynchronizedQueue**

```java
class BusySynchronizedQueue<E> implements BoundedQueue<E> {
    private ListedList<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 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 BoundedQueue<E> {
    private ListedList<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 can be active in any given object

    This constraint may actually be a liability for certain types of concurrently accessed objects, e.g., double-ended queues implemented as linked lists
```
Java synchronized methods protect critical sections from concurrent access. Adding the synchronized keyword has two effects.

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

    private ListedList<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(); }

    ...

See docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html
```
Java synchronized methods protects critical sections from concurrent access

- Adding the synchronized keyword has two effects

```java
class BusySynchronizedQueue<E> implements BoundedQueue<E> {
    private ListedList<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(); }
    ...
}
```

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

i.e., each synchronized method is “atomic?”
Java synchronized methods protects critical sections from concurrent access.

- Adding the synchronized keyword has two effects

```java
class BusySynchronizedQueue<E> implements BoundedQueue<E> {
    private ListedList<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; }
        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)
Evaluating the Busy SynchronizedQueue
Evaluating the BusySynchronizedQueue

- There are limitations with Java synchronized methods when they are used alone.

See [github.com/douglas craig schmidt/POSA/tree/master/ex/M3/Queues/BusySynchronizedQueue](https://github.com/douglas craig schmidt/POSA/tree/master/ex/M3/Queues/BusySynchronizedQueue)
Evaluating the BusySynchronizedQueue

- There are limitations with Java synchronized methods when they are used alone

```java
class BusySynchronizedQueue<E> implements BoundedQueue<E> {
    private LinkedList<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(); }

    ...
```

Concurrent calls to these methods will “busy wait”.

See [en.wikipedia.org/wiki/Busy_waiting](en.wikipedia.org/wiki/Busy_waiting)
Evaluating the BusySynchronizedQueue

- There are limitations with Java synchronized methods when they are used alone

```java
class BusySynchronizedQueue<E> implements BoundedQueue<E> {
    private LinkedList<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(); }
    ...
}
```

Need to coordinate `offer()` & `poll()` so they won’t busy wait when there’s nothing to do

Java built-in monitor objects therefore provide “wait” & “notify” mechanisms to avoid busy waiting
End of Java Monitor
Objects: Synchronization
(Part 3)