Java Monitor Objects:
Coordination Example Implementation

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 Module

• Learn how to fix a buggy concurrent Java program using Java’s wait & notify mechanisms, which provide *coordination*

• Visualize how Java monitor objects can be used to ensure mutual exclusion & coordination between threads running in a concurrent program

• Know how to program the Simple BlockingBoundedQueue in Java
Code Analysis of the SimpleBlockingBounded Queue Example
Code Analysis of SimpleBoundedBlockingQueue

• This class provides a simple synchronized blocking queue

Code Analysis of SimpleBoundedBlockingQueue

- This class provides a simple synchronized blocking queue.

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

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

This class provides a simple synchronized blocking queue

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

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

This internal state must be protected against race conditions

Code Analysis of SimpleBoundedBlockingQueue

- This class provides a simple synchronized blocking queue

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

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

The constructor need not be protected against race conditions. A constructor is only called once in one thread so there won’t be race conditions.
A thread can “wait” for a condition in a synchronized method.

```java
class SimpleBoundedBlockingQueue<E>
    implements SimpleBlockingQueue<E> {
    ...

    public synchronized String take() {
        while (isEmpty())
            wait();

        final E e = mList.poll();
        notifyAll();
        return e;
    }

    public synchronized Boolean isEmpty() {
        return mList.isEmpty();
    }
    ...
}
```

See en.wikipedia.org/wiki/Guarded_suspension
Code Analysis of SimpleBoundedBlockingQueue

- A thread can “wait” for a condition in a synchronized method

```java
class SimpleBoundedBlockingQueue<E> implements SimpleBlockingQueue<E>
{
    ...

    public synchronized String take()
    {
        while (isEmpty())
            wait();

        final E e = mList.poll();
        notifyAll();
        return e;
    }

    public synchronized Boolean isEmpty()
    {
        return mList.isEmpty();
    }

    ...
}
```

E.g., thread T₁ calls take(), which acquires the intrinsic lock & waits while the queue is empty
A thread can “wait” for a condition in a synchronized method.

```java
class SimpleBoundedBlockingQueue<E>
    implements SimpleBlockingQueue<E>
{
    ...

    public synchronized String take(){
        while (isEmpty())
            wait();

        final E e = mList.poll();

        notifyAll();
        return e;
    }

    public synchronized Boolean isEmpty(){
        return mList.isEmpty();
    }
}
...
```

*Check if the list is empty*
Code Analysis of SimpleBoundedBlockingQueue

• A thread can “wait” for a condition in a synchronized method

```java
class SimpleBoundedBlockingQueue<E> implements SimpleBlockingQueue<E> {
    ...

    public synchronized String take() {
        while (isEmpty())
            wait();

        final E e = mList.poll();

        notifyAll();
        return e;
    }

    public synchronized Boolean isEmpty() {
        return mList.isEmpty();
    }
    ...
}
```

The `isEmpty()` method is synchronized, demonstrating the Java monitor object “reentrant mutex” semantics

See en.wikipedia.org/wiki/Reentrant_mutex
Code Analysis of SimpleBoundedBlockingQueue

- `wait()` should be called in a loop that checks whether the condition is true or not.

```java
class SimpleBoundedBlockingQueue&lt;E&gt; implements SimpleBlockingQueue&lt;E&gt; {
    ...
    
    public synchronized String take() {
        while (isEmpty())
            wait();

        final E e = mList.poll();
        notifyAll();
        return e;
    }

    public synchronized Boolean isEmpty() {
        return mList.isEmpty();
    }
    ...
}
```

See docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html
• `wait()` should be called in a loop that checks whether the condition is true or not.
• A thread can’t assume a notification it receives is for *its* condition expression.

```java
class SimpleBoundedBlockingQueue<E> implements SimpleBlockingQueue<E> {
    ... 
    public synchronized String take() {
        while (isEmpty())
            wait();

        final E e = mList.poll();
        notifyAll();
        return e;
    }

    public synchronized Boolean isEmpty() {
        return mList.isEmpty();
    }
    ...
}
```

See [stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again/3186336#3186336](https://stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again/3186336#3186336)
Code Analysis of SimpleBoundedBlockingQueue

- `wait()` should be called in a loop that checks whether the condition is true or not.
- A thread can’t assume a notification it receives is for *its* condition expression.
- It also can’t assume the condition expression is true!

```java
class SimpleBoundedBlockingQueue<E> implements SimpleBlockingQueue<E>
{
    ...

    public synchronized String take(){
        while (isEmpty())
            wait();

        final E e = mList.poll();

        notifyAll();
        return e;
    }

    public synchronized Boolean isEmpty(){
        return mList.isEmpty();
    }

    ...
}
```

*i.e., due to the inherent nondeterminism of concurrency*

See [en.wikipedia.org/wiki/Nondeterministic_algorithm](en.wikipedia.org/wiki/Nondeterministic_algorithm)
Code Analysis of SimpleBoundedBlockingQueue

- `wait()` should be called in a loop that checks whether the condition is true or not.
- A thread can’t assume a notification it receives is for *its* condition expression.
- It also can’t assume the condition expression is true!
- Must also guard against “spurious wakeups”.
- A thread might be awoken in `wait()` even if no thread called `notify()`/`notifyAll()`.

```java
class SimpleBoundedBlockingQueue<E> implements SimpleBlockingQueue<E> {
    ...
    public synchronized String take() {
        while (isEmpty())
            wait();

        final E e = mList.poll();
        notifyAll();
        return e;
    }

    public synchronized Boolean isEmpty() {
        return mList.isEmpty();
    }
    ...
}
```

See [en.wikipedia.org/wiki/Spurious_wakeup](en.wikipedia.org/wiki/Spurious_wakeup)
Code Analysis of SimpleBoundedBlockingQueue

- A thread blocked on `wait()` won’t continue until it’s notified that the condition expression may be true

```java
class SimpleBoundedBlockingQueue<E> implements SimpleBlockingQueue<E> {
    ...

    public synchronized String take() {
        while (isEmpty())
            wait();

        final E e = mList.poll();
        notifyAll();
        return e;
    }

    public synchronized Boolean isEmpty() {
        return mList.isEmpty();
    }
    ...
```
Code Analysis of SimpleBoundedBlockingQueue

- A thread blocked on wait() won’t continue until it’s notified that the condition expression may be true.

```
public synchronized void put(E msg){
    while (isFull())
        wait();
    mList.add(msg);
    notifyAll();
}
```

```
private synchronized boolean isFull(){
    return mList.size() >= mCapacity;
}
```
A thread blocked on `wait()` won’t continue until it’s notified that the condition expression may be true.

Class `SimpleBoundedBlockingQueue` implements `SimpleBlockingQueue`.

```java
class SimpleBoundedBlockingQueue<E> implements SimpleBlockingQueue<E> {
    ...

    public synchronized void put(E msg) {
        ...
        while (isFull())
            wait();

        mList.add(msg);
        notifyAll();
    }

    private synchronized boolean isFull() {
        return mList.size() >= mCapacity;
    }
    ...
```
A thread blocked on `wait()` won’t continue until it’s notified that the condition expression may be true

```
thread T_2 calls `notifyAll()`, which will wakeup thread T_1 that’s blocking in `wait()`
```

```java
class SimpleBoundedBlockingQueue<E>
    implements SimpleBlockingQueue<E>
{
    ...

    public synchronized void put(E msg) {
        ...
        while (isFull())
            wait();

        mList.add(msg);
        notifyAll();
    }

    private synchronized boolean isFull() {
        return mList.size() >= mCapacity;
    }
    ...
```
• A thread blocked on `wait()` won’t continue until it’s notified that the condition expression may be true

```java
class SimpleBoundedBlockingQueue<E> implements SimpleBlockingQueue<E> {
  
  public synchronized void put(E msg) {
    while (isFull())
      wait();

    mList.add(msg);
    notifyAll();
  }

  private synchronized boolean isFull() {
    return mList.size() >= mCapacity;
  }

  ...  
```

Again, `notifyAll()` is used due to a Java monitor object only having a single wait queue..

See [stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again/3186336#3186336](http://stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again/3186336#3186336)
Code Analysis of SimpleBoundedBlockingQueue

- Several steps occur when a waiting thread is notified

```java
class SimpleBoundedBlockingQueue<E>
    implements SimpleBlockingQueue<E>
{
    ...

    public synchronized String take()
    {
        while (isEmpty())
            wait();

        notifyAll();
        return mList.poll();
    }
```
Several steps occur when a waiting thread is notified
- wakes up & obtains lock

Code Analysis of SimpleBoundedBlockingQueue

```java
class SimpleBoundedBlockingQueue<E> implements SimpleBlockingQueue<E> {
    ...

    public synchronized String take() {
        while (isEmpty())
            wait();

        notifyAll();
        return mList.poll();
    }
}
```
Code Analysis of SimpleBoundedBlockingQueue

- Several steps occur when a waiting thread is notified
  - wakes up & obtains lock
  - re-evaluates the condition expression

```java
class SimpleBoundedBlockingQueue<E> implements SimpleBlockingQueue<E> {
  ...

  public synchronized String take() {
    while (!isEmpty())
      wait();

    notifyAll();
    return mList.poll();
  }
```
Code Analysis of SimpleBoundedBlockingQueue

- Several steps occur when a waiting thread is notified:
  - wakes up & obtains lock
  - re-evaluates the condition expression
  - continues after wait()

```java
class SimpleBoundedBlockingQueue<E> implements SimpleBlockingQueue<E> {
    ...

    public synchronized String take() {
        while (isEmpty())
            wait();

        notifyAll();
        return mList.poll();
    }
}
```

*Calling `notifyAll()` before removing/returning the front item in the queue is ok since the monitor lock is held & only one method can be in monitor*
Several steps occur when a waiting thread is notified:
- wakes up & obtains lock
- re-evaluates the condition expression
- continues after wait()
- releases lock when it returns

```java
class SimpleBoundedBlockingQueue<E>
    implements SimpleBlockingQueue<E>
{
    ...

    public synchronized String take(){
        while (isEmpty())
            wait();

        notifyAll();
        return mList.poll();
    }
}
End of Java Monitor Objects: Coordination Example Implementation