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 that limited to a given # of elements

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;
    }

    ...
```
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 internal state must be protected against race conditions
This class provides a simple synchronized blocking queue.

Class SimpleBoundedBlockingQueue

```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](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 $T_1$ 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.
• 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();
    }
    ...
```

*isEmpty() is synchronized via the Java monitor object “reentrant mutex” semantics*
• wait() should be called in a loop that checks whether the condition is true or not

```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 docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html
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.

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

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

        final E e = mList.poll();

        notifyAll();
        return e;
    }
    ...

    See 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;
    }
    ...
}
```

*i.e., due to the inherent non-determinism of concurrency*
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;
    }

    ...
}
```

See [en.wikipedia.org/wiki/Spurious_wakeup](en.wikipedia.org/wiki/Spurious_wakeup)
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;
    }
    ...
}
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 void put(E msg) {
        ...
        while (isFull())
            wait();
        mList.add(msg);
        notifyAll();
    }

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

*e.g., thread T₂ calls put(), which acquires the intrinsic lock & adds an item to the queue so it’s no longer empty.*
A thread blocked on `wait()` won’t continue until it’s notified that the condition expression may be true.

Assuming that thread $T_1$ is blocked in `take()` the queue won’t be full!
A thread blocked on `wait()` won’t continue until it’s notified that the condition expression may be true.

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

Thread T₂ calls `notifyAll()`, which will wakeup thread T₁ that’s blocking in `wait()`.

```java
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](https://stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again/3186336#3186336)
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

```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
- re-evaluates the condition expression

```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
- 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*
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()
  - releases lock when it returns

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

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

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