Android Concurrency & Synchronization: Part 4

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

CS 282 Principles of Operating Systems II
Systems Programming for Android
Learning Objectives in this Part of the Module

• Understand the Android mechanisms available to implement concurrent apps that synchronize & schedule their interactions.

Diagram:

- **Producer**
  - put()

- **Synchronized Queue**
  - put()
  - take()

- **Consumer**
  - take()
Motivating Java Synchronization & Scheduling

- Consider a concurrent producer/consumer portion of a Java app.

Concurrent threads can corrupt the queue’s internal state if it is not synchronized properly.

Likewise, threads will ‘busy wait’ when the queue is empty or full, which wastes CPU cycles unnecessarily.
Motivating Java Synchronization & Scheduling

• Consider a concurrent producer/consumer portion of a Java app

• Here’s some example code that demonstrates the problem

```java
public class SynchronizedQueue {
    private List<String> q_ =
        new ArrayList<String>();

    public void put(String msg){ q_.add(msg); }
    public String take(){ return q_.remove(0); }

    public static void main(String argv[]){
        new Thread(new Runnable(){
            public void run(){
                for(int i = 0; i < 10; i++) put(Integer.toString(i));
            }).start();
        new Thread(new Runnable(){
            public void run(){
                while(true){ System.out.println(take());}
            }).start();
    
```

```java```
Motivating Java Synchronization & Scheduling

• Consider a concurrent producer/consumer portion of a Java app
• Here’s some example code that demonstrates the problem

```java
public class SynchronizedQueue {
    private List<String> q_ = new ArrayList<String>();

    public void put(String msg) { q_.add(msg); }
    public String take() { return q_.remove(0); }

    public static void main(String argv[]) {
        new Thread(new Runnable() {
            public void run() {
                for(int i = 0; i < 10; i++) put(Integer.toString(i));
            }
        }).start();
        new Thread(new Runnable() {
            public void run() {
                while(true) { System.out.println(take()); }
            }
        }).start();
    }
}
```

Resizable-array implementation

Motivating Java Synchronization & Scheduling
Motivating Java Synchronization & Scheduling

• Consider a concurrent producer/consumer portion of a Java app
• Here’s some example code that demonstrates the problem

```java
public class SynchronizedQueue {
    private List<String> q_ =
        new ArrayList<String>();

    public void put(String msg){ q_.add(msg); }
    public String take(){ return q_.remove(0); }

    public static void main(String argv[]) {
        new Thread(new Runnable()
            public void run(){
                for(int i = 0; i < 10; i++) put(Integer.toString(i));
            }).start();
        new Thread(new Runnable()
            public void run()
                while(true){ System.out.println(take()); }
            ).start();
    }
```

Enqueue & dequeue strings into/from the queue
Motivating Java Synchronization & Scheduling

• Consider a concurrent producer/consumer portion of a Java app

• Here’s some example code that demonstrates the problem

```java
public class SynchronizedQueue {
    private List<String> q_ =
        new ArrayList<String>();

    public void put(String msg){ q_.add(msg); }
    public String take(){ return q_.remove(0); }

    public static void main(String argv[]) {
        new Thread(new Runnable(){
            public void run(){
                for(int i = 0; i < 10; i++) put(Integer.toString(i));
            }).start();
        new Thread(new Runnable(){
            public void run(){
                while(true){ System.out.println(take());}
            }).start();
```
Motivating Java Synchronization & Scheduling

- Consider a concurrent producer/consumer portion of a Java app
- Here's some example code that demonstrates the problem

```java
public class SynchronizedQueue {
    private List<String> q_ =
        new ArrayList<String>();

    public void put(String msg) { q_.add(msg); }
    public String take() { return q_.remove(0); }

    public static void main(String argv[]) {
        new Thread(new Runnable(){
            public void run(){
                for(int i = 0; i < 10; i++) put(Integer.toString(i));
            }).start();
        new Thread(new Runnable(){
            public void run(){
                while(true){ System.out.println(take());}
            }).start();
    }
```
Motivating Java Synchronization & Scheduling

- Consider a concurrent producer/consumer portion of a Java app
- Here’s some example code that demonstrates the problem

```java
public class SynchronizedQueue {
    private List<String> q_ =
        new ArrayList<String>();

    public void put(String msg) { q_.add(msg); }
    public String take() { return q_.remove(0); }

    public static void main(String argv[]) {
        new Thread(new Runnable() {
            public void run() {
                for (int i = 0; i < 10; i++) put(Integer.toString(i));
            }
        }).start();
        new Thread(new Runnable() {
            public void run() {
                while (true) { System.out.println(take()); }
            }
        }).start();
    }
}
```

Must protect critical sections from being run by two threads concurrently
Partial Solution Using Java Synchronization

- Java provides the “synchronized” keyword to specify sections of code in an object that cannot be accessed concurrently by two threads.

```java
public class SynchronizedQueue {
    private List<String> q_ =
        new ArrayList<String>();

    public void synchronized put(String msg){ q_.add(msg); }
    public String synchronized take(){ return q_.remove(0); }

    public static void main(String argv[])
    {
        new Thread(new Runnable(){
            public void run()
            {
                for(int i = 0; i < 10; i++) put(Integer.toString(i));
            }).start();

        new Thread(new Runnable(){
            public void run()
            {
                while(true){ System.out.println(take());}
            }).start();
    }
```

Only one synchronized method can be active in any given object.

There are still problems with this solution...
Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects

```java
public class SynchronizedQueue {
    private List<String> q_ = new ArrayList<String>();

    public synchronized void put(String msg) {
        q_.add(msg);
        notifyAll();
    }

    public synchronized String take() {
        while (q_.isEmpty()) {
            wait();
        }
        ... return q_.remove(0);
    }
}
```
Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects
- Methods requiring mutual exclusion must be explicitly marked with the synchronized keyword

```java
public class SynchronizedQueue {
    private List<String> q_ =
        new ArrayList<String>();

    public synchronized void put(String msg) {
        q_.add(msg);
        notifyAll();
    }

    public synchronized String take() {
        while (q_.isEmpty()) {
            wait();
        }
        ...
        return q_.remove(0);
    }
}
```

See [docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html](http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html)
Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects
  - Methods requiring mutual exclusion must be explicitly marked with the synchronized keyword
- Access to a synchronized method is serialized w/other synchronized methods
Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects
- Java also supports synchronized blocks
  - e.g.,
    ```java
    public class SynchronizedQueue {
        private List<String> q_ =
            new ArrayList<String>();

        public synchronized void put(String msg) {
            ...
            synchronized (this) {
                q_.add(msg);
                notifyAll();
            }
        }

        public synchronized String take() {
            while (q_.isEmpty()) {
                wait();
            }
            return q_.remove(0);
        }
    }
    ```
Android Concurrency & Synchronization

Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects
- Java also supports synchronized blocks
  - e.g.,
    ```java
    void put(String msg)
    {
      ...
      synchronized (this) {
        q_.add(msg);
        notifyAll();
      }
    }
    ```
  - Synchronized blocks enable more fine-grained serialization

```java
public class SynchronizedQueue {
  private List<String> q_ =
      new ArrayList<String>();

  public synchronized void put(String msg){
    ...
    q_.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (q_.isEmpty()) {
      wait();
    }
    ...
    return q_.remove(0);
  }
}
```

docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html has more
Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects
- Java also supports synchronized blocks
- Java objects have wait() & notify()/notifyAll() methods that allow callers to wait for a condition to become true

```java
public class SynchronizedQueue {
    private List<String> q_ =
        new ArrayList<String>();

    public synchronized void put(String msg){
        q_.add(msg);
        notifyAll();
    }

    public synchronized String take(){
        while (q_.isEmpty()) {
            wait();
        }
        return q_.remove(0);
    }
}
```
Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects
- Java also supports synchronized blocks
- Java objects have wait() & notify()/notifyAll() methods that allow callers to wait for a condition to become true
- Calling wait() on an object will suspend current thread until a notify*() call is made on the same object

```java
public class SynchronizedQueue {
    private List<String> q_ =
        new ArrayList<String>();

    public synchronized void put(String msg) {
        ...
        q_.add(msg);
        notifyAll();
    }

    public synchronized String take() {
        while (q_.isEmpty()) {
            wait();
        }
        ...
        return q_.remove(0);
    }
}
```
Better Solution Using Java Monitor Objects

- All objects in Java can be Monitor Objects
- Java also supports synchronized blocks
- Java objects have wait() & notify()/notifyAll() methods that allow callers to wait for a condition to become true
- Calling wait() on an object will suspend current thread until a notify*() call is made on the same object
- Calling notifyAll() will wake up all waiting threads

```java
public class SynchronizedQueue {
    private List<String> q_ = new ArrayList<String>();

    public synchronized void put(String msg){
        ... 
        q_.add(msg);
        notifyAll();
    }

    public synchronized String take(){
        while (q_.isEmpty()) {
            wait();
        }
        ...
        return q_.remove(0);
    }
}
```

[stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again](https://stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again)
Detailed Analysis of wait() & notifyAll()

• Inside a synchronized method, you can request a thread “wait” for a condition, e.g.:

• The synchronized take() method acquires the monitor lock, checks the queue size, & waits if the queue is empty.

```java
class SynchronizedQueue {
    private List<String> q_ = new ArrayList<String>();

    public synchronized void put(String msg) {
        q_.add(msg);
        notifyAll();
    }

    public synchronized String take() {
        while (q_.isEmpty()) {
            wait();
        }
        return q_.remove(0);
    }
}
```
Detailed Analysis of wait() & notifyAll()

- Inside a synchronized method, you can request a thread “wait” for a condition, e.g.:

- The synchronized `take()` method acquires the monitor lock, checks the queue size, & waits if the queue is empty

```java
public class SynchronizedQueue {
    private List<String> q_ = new ArrayList<String>();

    public synchronized void put(String msg)
    {
        ...
        q_.add(msg);
        notifyAll();
    }

    public synchronized String take()
    {
        while (q_.isEmpty())
        {
            wait();
        }
        ...
        return q_.remove(0);
    }
}
```

- Always invoke `wait()` inside a loop that tests for the condition being waited for

- Don't assume the notification was for the particular condition being waited for or that the condition is still true

See docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html
Detailed Analysis of wait() & notifyAll()

- Inside a synchronized method, you can request a thread “wait” for a condition, e.g.:
  - The synchronized take() method acquires the monitor lock, checks the queue size, & waits if the queue is empty
  - The thread blocking on wait() doesn’t continue until another thread notifies it that the queue has data to process

```java
class SynchronizedQueue {
    private List<String> q_ = new ArrayList<String>();

    public synchronized void put(String msg) {
        ... 
        q_.add(msg);
        notifyAll();
    }

    public synchronized String take() {
        while (!q_.isEmpty()) { 
            wait();
        }
        ... 
        return q_.remove(0);
    }
}
```
Inside a synchronized method, you can request a thread “wait” for a condition, e.g.:

- The synchronized take() method acquires the monitor lock, checks the queue size, & waits if the queue is empty.
- The thread blocking on wait() doesn’t continue until another thread notifies it that the queue has data to process.
- When the thread is notified, it wakes up, obtains the monitor lock, continues after the wait() call, & releases the lock when the method returns.

```java
public class SynchronizedQueue {
    private List<String> q_ = new ArrayList<String>();

    public synchronized void put(String msg) {
        q_.add(msg);
        notifyAll();
    }

    public synchronized String take() {
        while (q_.isEmpty()) {
            wait();
        }
        return q_.remove(0);
    }
}
```
Summary

- Each Java object may be used as a monitor object
Summary

- Each Java object may be used as a monitor object
- Methods requiring mutual exclusion must be explicitly marked with the synchronized keyword
Summary

• Each Java object may be used as a monitor object
  • Methods requiring mutual exclusion must be explicitly marked with the synchronized keyword
• Blocks of code may also be marked by synchronized

```java
void put(String msg) {
    synchronized (this) {
        q_.add(msg);
        notifyAll();
    }
    ...
}
```

Producer

```
<table>
<thead>
<tr>
<th>put()</th>
</tr>
</thead>
</table>
```

Synchronized Queue

```
synchronized put()
synchronized take()
```

Consumer

```
take() |
|------|
```

docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html has more
Summary

- Each Java object may be used as a monitor object.
- Each monitor object in Java is equipped with a single wait queue in addition to its entrance queue.
- All waiting is done on this single wait queue & all notify() & notifyAll() operations apply to this queue.

en.wikipedia.org/wiki/Monitor_(synchronization)#Implicit_condition_variable_monitors
Summary

- Production Java apps may need more than the simply monitor mechanisms

<table>
<thead>
<tr>
<th>Interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Condition</strong></td>
</tr>
<tr>
<td><strong>Lock</strong></td>
</tr>
<tr>
<td><strong>ReadWriteLock</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AbstractOwnableSynchronizer</strong></td>
</tr>
<tr>
<td><strong>AbstractQueuedLongSynchronizer</strong></td>
</tr>
<tr>
<td><strong>AbstractQueuedLongSynchronizer.ConditionObject</strong></td>
</tr>
<tr>
<td><strong>AbstractQueuedSynchronizer</strong></td>
</tr>
<tr>
<td><strong>AbstractQueuedSynchronizer.ConditionObject</strong></td>
</tr>
<tr>
<td><strong>LockSupport</strong></td>
</tr>
<tr>
<td><strong>ReentrantLock</strong></td>
</tr>
<tr>
<td><strong>ReentrantReadWriteLock</strong></td>
</tr>
<tr>
<td><strong>ReentrantReadWriteLock.ReadLock</strong></td>
</tr>
<tr>
<td><strong>ReentrantReadWriteLock.WriteLock</strong></td>
</tr>
</tbody>
</table>
Learning Objectives in this Part of the Module

- Understand the *Monitor Object* pattern & how it can be used to synchronize & schedule concurrent Android programs.
Monitor Object

**Intent**

- Synchronizes concurrent method execution to ensure only one method at a time runs within an object.
- Allows an object’s methods to cooperatively schedule their execution sequences.

See [www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf) for Monitor Object.
Applicability

• When an object’s interface methods should define its synchronization boundaries
Monitor Object

**Applicability**

- When an object’s interface methods should define its synchronization boundaries
- When only one method at a time should be active within the same object
Applicability

- When an object’s interface methods should define its synchronization boundaries
- When only one method at a time should be active within the same object
- When objects should be responsible for method synchronization transparently, without requiring explicit client intervention
Monitor Object

POSA2 Concurrency

Applicability

- When an object’s interface methods should define its synchronization boundaries
- When only one method at a time should be active within the same object
- When objects should be responsible for method synchronization transparently, without requiring explicit client intervention
- When an object’s methods may block during their execution
Monitor Object  POSA2 Concurrency

Structure & Participants

- Monitor Object
  - sync method 1
  - ... (three dots)
  - sync method N

- Client
  - 2..* (relationship between Client and Monitor Object)

- Monitor Condition
  - wait()
  - notify()
  - notifyAll()

- Monitor Lock
  - acquire()
  - release()

Synchronized Queue
Monitor Object

POSAS2 Concurrency

Structure & Participants

Client

Monitor Object

- sync method 1
- ... (omitted)
- sync method N

Monitor Condition

- wait()
- notify()
- notifyAll()

Monitor Lock

- acquire()
- release()

Java Monitor Lock
Monitor Object                  POSA2 Concurrency

Structure & Participants

Note that Java monitor objects only have a single (implicit) monitor condition
Monitor Object

Dynamics

Synchronized method invocation & serialization
Dynamics

Monitor Object

POSA2 Concurrency

Synchronized method thread suspension
Monitor Object

POSAS2 Concurrency

Dynamics

Monitor condition notification

the OS thread scheduler automatically suspends the client thread

sync_method1()  
acquire()  
dowork()  
wait()  
notity()  

the OS thread scheduler automatically resumes the client thread and the synchronized method

sync_method2()  
acquire()  
dowork()  
release()  
notify()  

the OS thread scheduler atomically releases the monitor lock

the OS thread scheduler atomically reacquires the monitor lock

D. C. Schmidt

Android Concurrency & Synchronization
Monitor Object  POSA2 Concurrency

Dynamics

- **Synchronized method thread resumption**

  - The OS thread scheduler automatically suspends the client thread.
  - The OS thread scheduler automatically resumes the client thread and the synchronized method.

  - The OS thread scheduler atomically acquires the monitor lock.
  - The OS thread scheduler atomically reacquires the monitor lock.

  - Acquire
  - Release
  - Notify
  - Wait
  - DoWork
  - Sync method1()
Monitor Object example in Android

- The CancellationSignal class provides the ability to cancel an operation that's in progress.

```java
public final class CancellationSignal {
    private boolean mCancelInProgress;

    public void setOnCancelListener (OnCancelListener listener) {
        ...
    }

    public void cancel() {
        ...
    }
}
```

See developer.android.com/reference/android/os/CancellationSignal.html
Monitor Object example in Android

- The CancellationSignal class provides the ability to cancel an operation that's in progress.
- Used for long-running operations like ContentResolver.query().

```java
public final class CancellationSignal {
    private boolean mCancelInProgress;

    public void setOnCancelListener(OnCancelListener listener) {
        ...
    }

    public void cancel() {
        ...
    }

    ...
}
```

Monitor Object example in Android

- The CancellationSignal class provides the ability to cancel an operation that’s in progress
- Several method are used to implement Monitor Object
- setOnCancelListener() – Sets the cancellation listener whose onCancel() hook will be called when an operation is cancelled

```java
public final class CancellationSignal {
    private boolean mCancelInProgress;

    public void setOnCancelListener(OnCancelListener listener) {
        synchronized (this) {
            while (mCancelInProgress) {
                try { wait(); } catch (InterruptedException ex) {}
            }
            mOnCancelListener = listener;
        }
    }
}

frameworks/base/core/java/android/os/CancellationSignal.java has the code
Monitor Object example in Android

- The CancellationSignal class provides the ability to cancel an operation that’s in progress.
- Several method are used to implement Monitor Object:
  - setOnCancelListener() – Sets the cancellation listener whose onCancel() hook will be called when an operation is cancelled.
  - cancel() – Cancels operation & signals cancellation listener.

```java
public final class CancellationSignal {
    ...
    public void cancel() {
        synchronized (this) {
            mCancelInProgress = true;
            ...
        }
        try {
            ...
            listener.onCancel();
            ...
        } finally {
            synchronized (this) {
                mCancelInProgress = false;
                notifyAll();
            }
        }
    }
}
```

Frameworks/base/core/java/android/os/CancellationSignal.java has the code.
**Consequences**

+ Simplification of concurrency control
  
  - Presents a concise programming model for sharing an object among cooperating threads where object synchronization corresponds to method invocations.
Consequences

+ Simplification of concurrency control
+ Simplification of scheduling method execution

- Synchronized methods use their monitor conditions to determine the circumstances under which they should suspend or resume their execution & those of collaborating monitor objects

```java
public synchronized String take()
{
    while (q_.isEmpty())
    {
        wait();
    }
    ... return q_.remove(0);
}

public synchronized void put(String msg)
{
    ... q_.add(msg);
    notifyAll();
}
```
Consequences

- Limited Scalability

  A single monitor lock can limit scalability due to increased contention when multiple threads serialize on a monitor object.
Monitor Object

Consequences

- Limited Scalability
- Complicated extensibility semantics

• These result from the coupling between a monitor object’s functionality & its synchronization mechanisms

```java
public synchronized void put(String msg) {
    ...
    q_.add(msg);
    notifyAll();
}
```

```java
public synchronized String take() {
    while (q_.isEmpty()) {
        wait();
    }
    ...
    return q_.remove(0);
}
```

 wisnesky.net/anomaly.pdf has info on the “inheritance anomaly problem”
Monitor Object

Consequences

− Limited Scalability
− Complicated extensibility semantics
− Nested monitor lockout

• This problem can occur when monitor objects are nested

```java
class Inner {
    protected boolean cond_ = false;
    public synchronized void
        awaitCondition() {
            while (!cond)
                try { wait (); } catch
                    (InterruptedException e) {} 
        }
    public synchronized void
        signalCondition(boolean c) { 
            cond_ = c; notifyAll ();
        }
}

class Outer {
    protected Inner inner_ = 
        new Inner ();
    public synchronized void
        process() { 
            inner_.awaitCondition ();
        }
    public synchronized void
        set(boolean c) { 
            inner_.signalCondition (c);
        }
}
```

Holds the monitor lock

Method won’t execute!
### Monitor Object

**Implementation**

- Define the monitor object’s interface methods
- e.g., `ArrayBlockingQueue` is a bounded `BlockingQueue` backed by an array that queues elements in FIFO order

```java
public class ArrayBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>,
    java.io.Serializable {

    public void put(E e) ...

    public E take() ...

}
```

Monitor Object

Implementation

- Define the monitor object’s interface methods
- Define the monitor object’s implementation methods
- See the *Thread-Safe Interface* pattern for design rationale

POSA2 Concurrency

```java
public class ArrayBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>,
    java.io.Serializable {

    public void put(E e) ...

    public E take() ...

    private void insert(E x) ...

    private E extract() ...

}

www.dre.vanderbilt.edu/~schmidt/PDF/locking-patterns.pdf has more info
### Monitor Object

#### Implementation

- Define the monitor object’s interface methods
- Define the monitor object’s implementation methods
- Define the monitor object’s internal state & synchronization mechanisms
- Can use classes defined in the java.util.concurrent package

```java
public class ArrayBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>,
    java.io.Serializable {

    ... 

    final Object[] items;
    int takeIndex;
    int putIndex;
    int count;

    final ReentrantLock lock;
    private final Condition notEmpty;
    private final Condition notFull;
}
```

Implementation

- Define the monitor object’s interface methods
- Define the monitor object’s implementation methods
- Define the monitor object’s internal state & synchronization mechanisms
- Implement all the monitor object’s methods & data members
  - Note the Java synchronized keyword isn’t used here!

```java
public class ArrayBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>,
    java.io.Serializable {

    public void put(E e) throws InterruptedException {
        final ReentrantLock lock = this.lock;
        lock.lockInterruptibly();
        try {
            while (count == items.length)
                notFull.await();
            insert(e);
        } finally {
            lock.unlock();
        }
    }

    ...
```

libcore/luni/src/main/java/java/util/concurrent/ArrayBlockingQueue.java
Monitor Object

Implementation

- Define the monitor object’s interface methods
- Define the monitor object’s implementation methods
- Define the monitor object’s internal state & synchronization mechanisms
- Implement all the monitor object’s methods & data members
- Note the Java synchronized keyword isn’t used here!

```java
public class ArrayBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>,
    java.io.Serializable {

    public E take() throws InterruptedException {
        final ReentrantLock lock =
            this.lock;
        lock.lockInterruptibly();
        try {
            while (count == 0)
                notEmpty.await();
            return extract();
        } finally {
            lock.unlock();
        }
    }

    ...

    public class ArrayBlockingQueue.java
```
Monitor Object

Known Uses

- Dijkstra & Hoare-style Monitors

---

en.wikipedia.org/wiki/Monitor_(synchronization) describes monitors
Known Uses

- Dijkstra & Hoare-style Monitors
- Java objects with synchronized methods/blocks
- Note how few synchronized methods/blocks are used in java.util.concurrent, yet this pattern is still widely applied.

```java
public class ArrayBlockingQueue<E>
    extends AbstractQueue<E>
    implements BlockingQueue<E>,
    java.io.Serializable {

    public E take() throws InterruptedException {
        final ReentrantLock lock =
            this.lock;
        lock.lockInterruptibly();
        try {
            while (count == 0)
                notEmpty.await();
            return extract();
        } finally {
            lock.unlock();
        }
    }

    ...
```

[docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html has more](https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html)
Known Uses

- Dijkstra & Hoare-style Monitors
- Java objects with synchronized methods/blocks
- Android CancellationSignal

```java
public final class CancellationSignal {
    ... 
    private boolean mCancelInProgress;

    public void setOnCancelListener (OnCancelListener listener) {
        ...
    }

    public void cancel() {
        ...
    }
}
```

`frameworks/base/core/java/android/os/CancellationSignal.java` has the code
**Known Uses**

- Dijkstra & Hoare-style Monitors
- Java objects with synchronized methods/blocks
- Android CancellationSignal
- ACE provides portable C++ building blocks for implementing monitor objects

### ACE Class

<table>
<thead>
<tr>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACE_Guard</td>
</tr>
<tr>
<td>ACE_Read_Guard</td>
</tr>
<tr>
<td>ACE_Write_Guard</td>
</tr>
<tr>
<td>ACE_Thread_Mutex</td>
</tr>
<tr>
<td>ACE_Process_Mutex</td>
</tr>
<tr>
<td>ACE_Null_Mutex</td>
</tr>
<tr>
<td>ACE_RW_Thread_Mutex</td>
</tr>
<tr>
<td>ACE_RW_Process_Mutex</td>
</tr>
<tr>
<td>ACE_Thread_Semaphore</td>
</tr>
<tr>
<td>ACE_Process_Semaphore</td>
</tr>
<tr>
<td>ACE_Null_Semaphore</td>
</tr>
<tr>
<td>ACE_Condition_Thread_Mutex</td>
</tr>
<tr>
<td>ACE_Null_Condition</td>
</tr>
</tbody>
</table>

[www.dre.vanderbilt.edu/~schmidt/PDF/ACE-concurrency.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/ACE-concurrency.pdf) has more info
- Concurrent software often contains objects whose methods are invoked by multiple client threads
Summary

Concurrent software often contains objects whose methods are invoked by multiple client threads.

- To protect the internal state of shared objects, it is necessary to synchronize & schedule client access to them.

Monitor object

- A client invokes method
- Synchronization mechanism
- Client-thread-specific monitor object instances
- A client thread

block until object becomes available
Concurrent software often contains objects whose methods are invoked by multiple client threads.

To protect the internal state of shared objects, it is necessary to synchronize & schedule client access to them.

To simplify programming, however, clients should not need to distinguish programmatically between accessing shared & non-shared components.
Summary

- Concurrent software often contains objects whose methods are invoked by multiple client threads.

- The *Monitor Object* pattern enables the sharing of object by client threads that self-coordinate a serialized—yet interleaved—execution sequence.

See [www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf) for *Monitor Object*. 