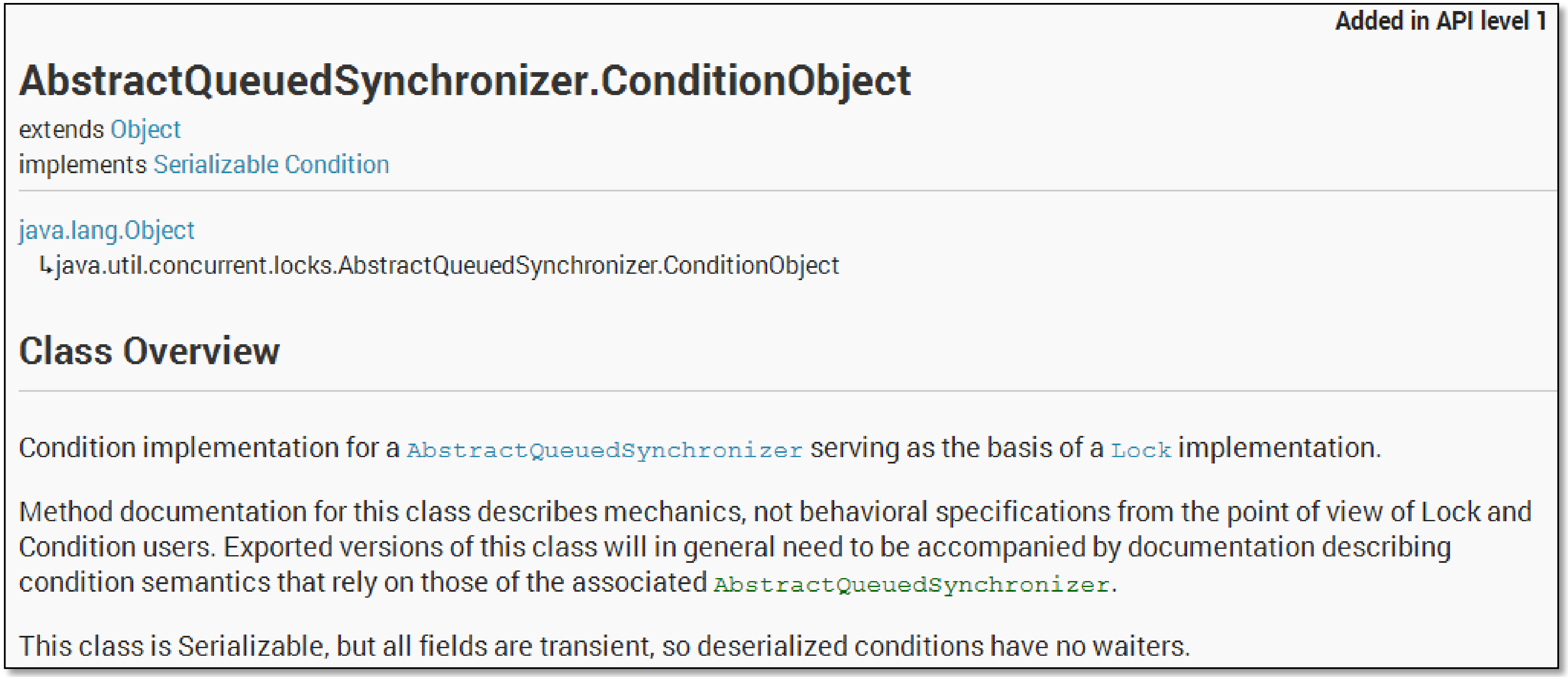
Android Concurrency: Java ConditionObject

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

- Understand how ConditionObjects enable user-defined Java objects to have multiple wait-sets per object



Added in API level 1

AbstractQueuedSynchronizer.ConditionObject

extends Object
implements Serializable Condition

java.lang.Object
↳java.util.concurrent.locks.AbstractQueuedSynchronizer.ConditionObject

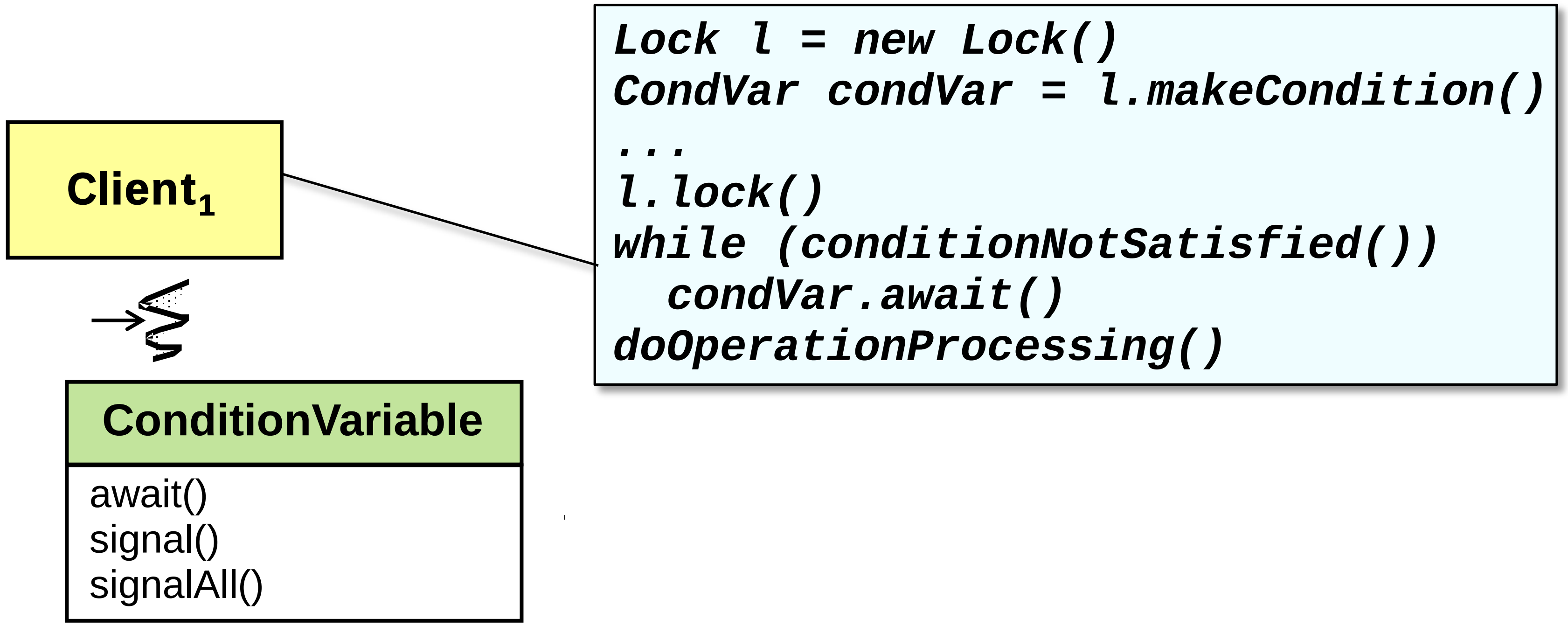
Class Overview

Condition implementation for a `AbstractQueuedSynchronizer` serving as the basis of a `Lock` implementation.

Method documentation for this class describes mechanics, not behavioral specifications from the point of view of Lock and Condition users. Exported versions of this class will in general need to be accompanied by documentation describing condition semantics that rely on those of the associated `AbstractQueuedSynchronizer`.

This class is Serializable, but all fields are transient, so deserialized conditions have no waiters.

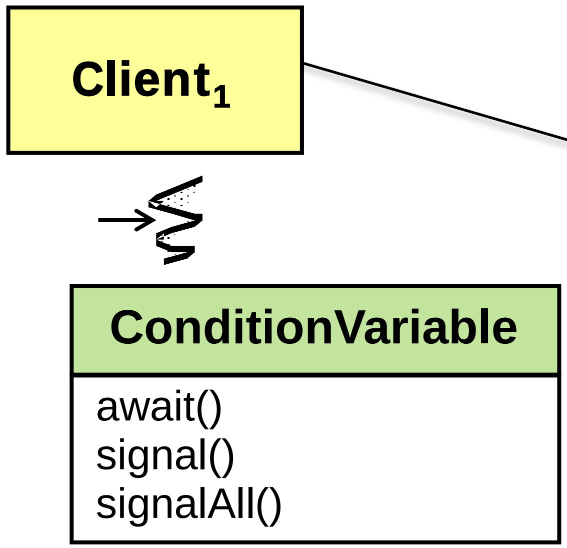
Overview of Condition Variables



```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

- A condition variable is used to implement the *Guarded Suspension* pattern

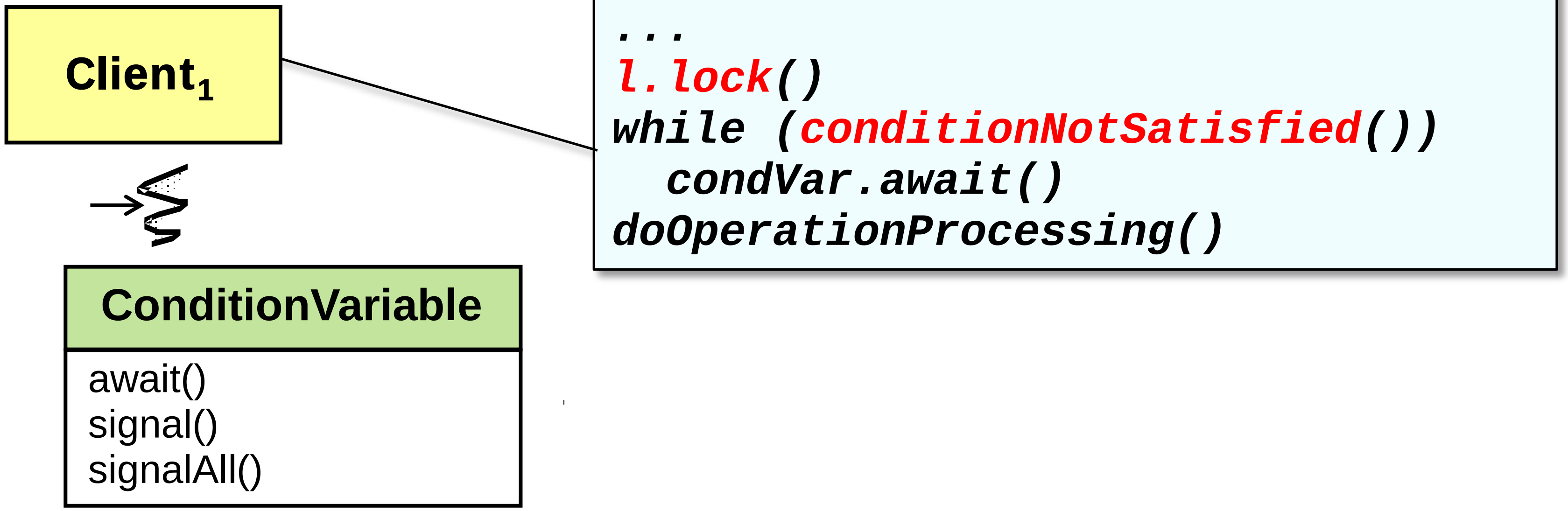
Overview of Condition Variables



```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

- A condition variable is used to implement the *Guarded Suspension* pattern

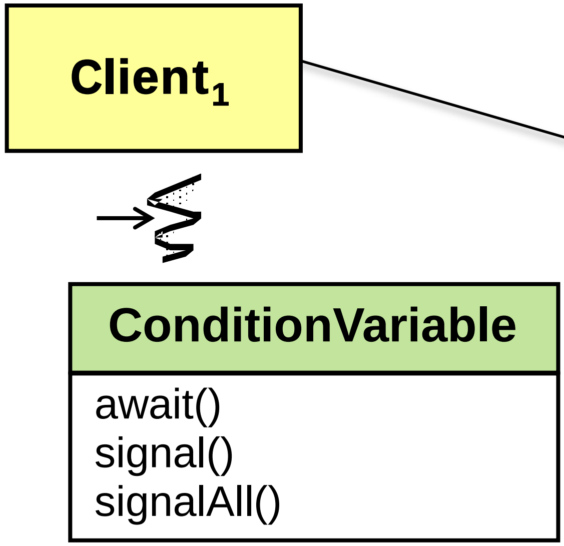
Overview of Condition Variables



```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

- A condition variable is used to implement the *Guarded Suspension* pattern

Overview of Condition Variables

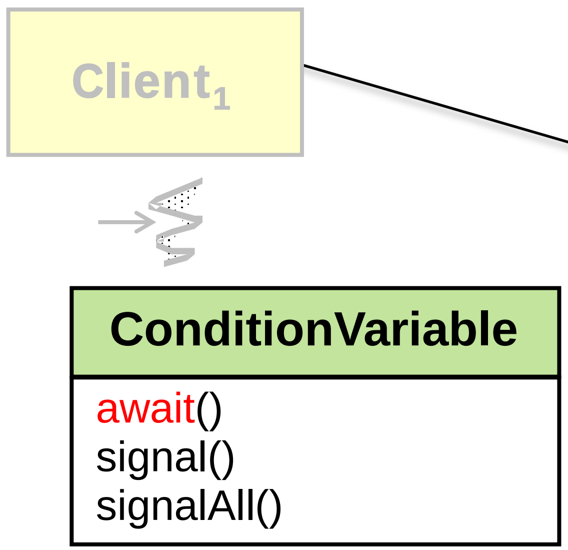


```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

- A condition variable is used to implement the *Guarded Suspension* pattern

en.wikipedia.org/wiki/Guarded_suspension has more on *Guarded Suspension*

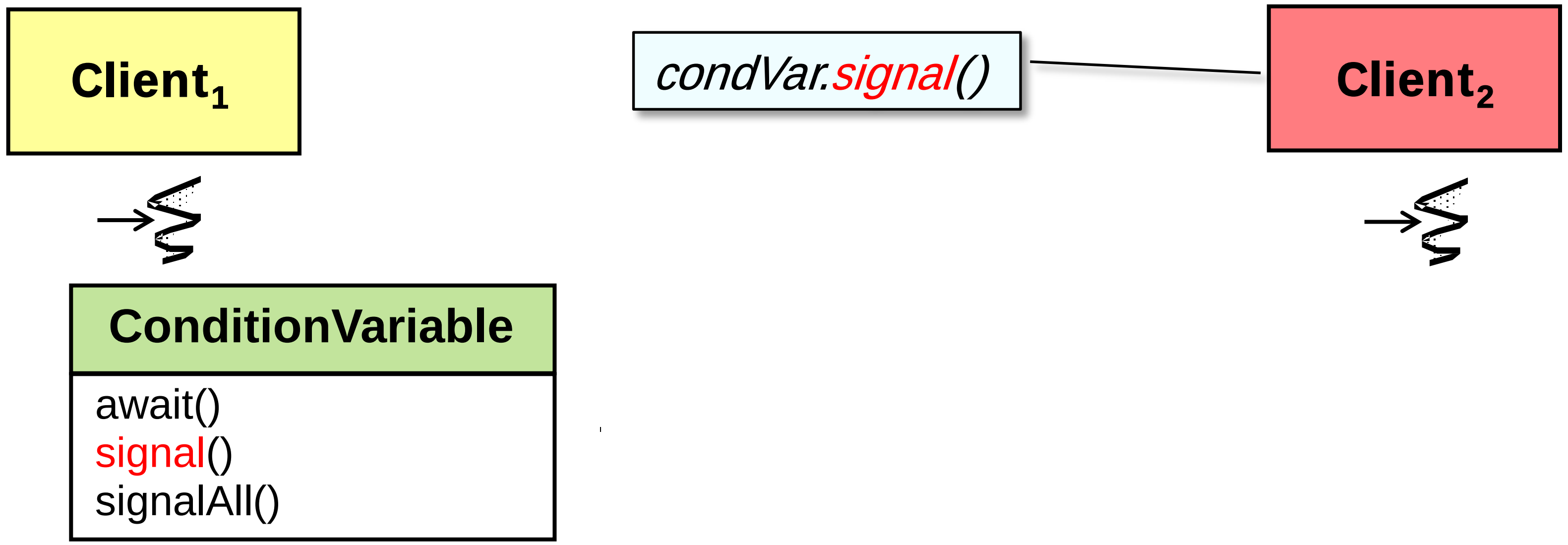
Overview of Condition Variables



```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

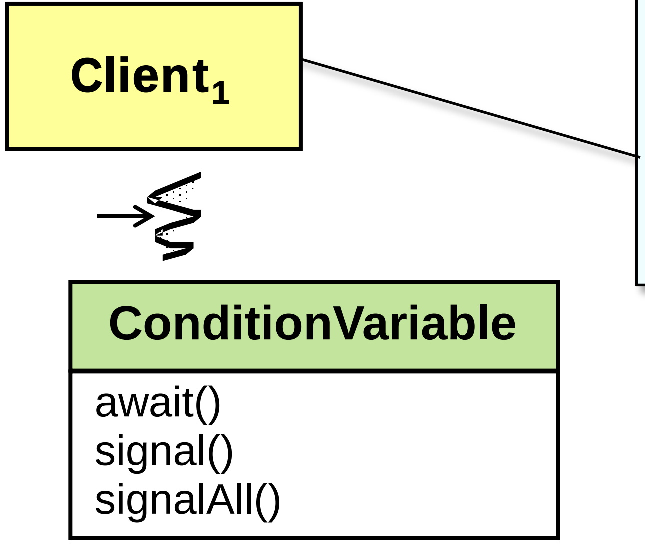
- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true

Overview of Condition Variables



- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true

Overview of Condition Variables

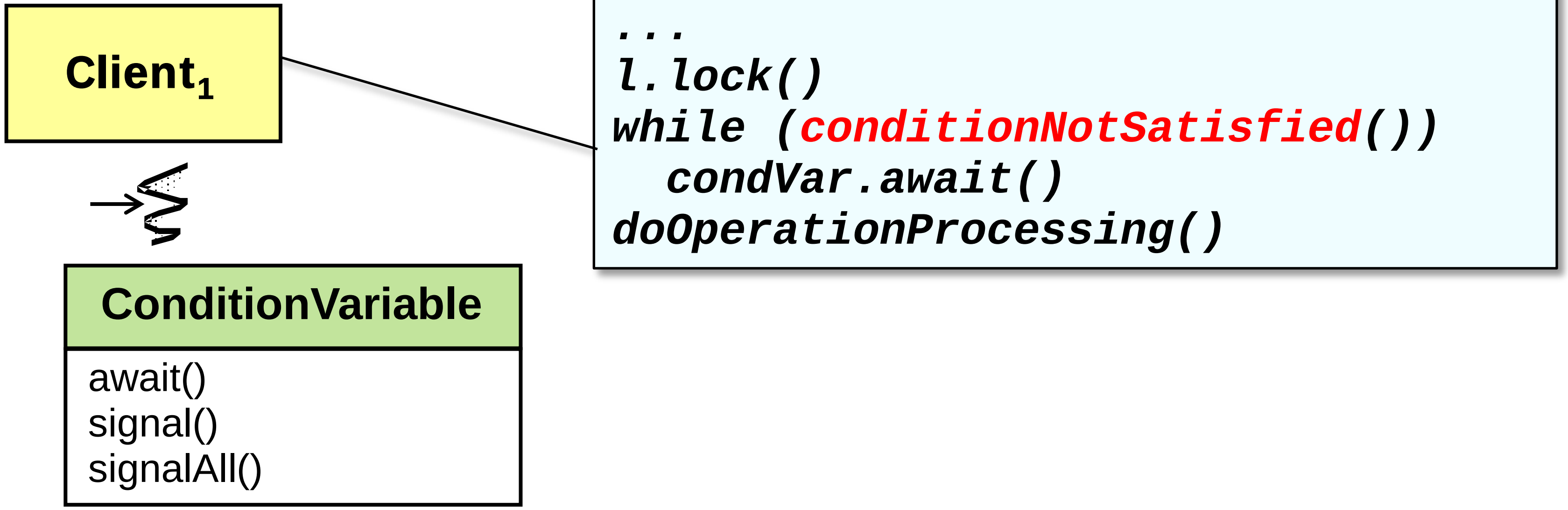


```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true

See en.wikipedia.org/wiki/Monitor_(synchronization)#Condition_variables_2

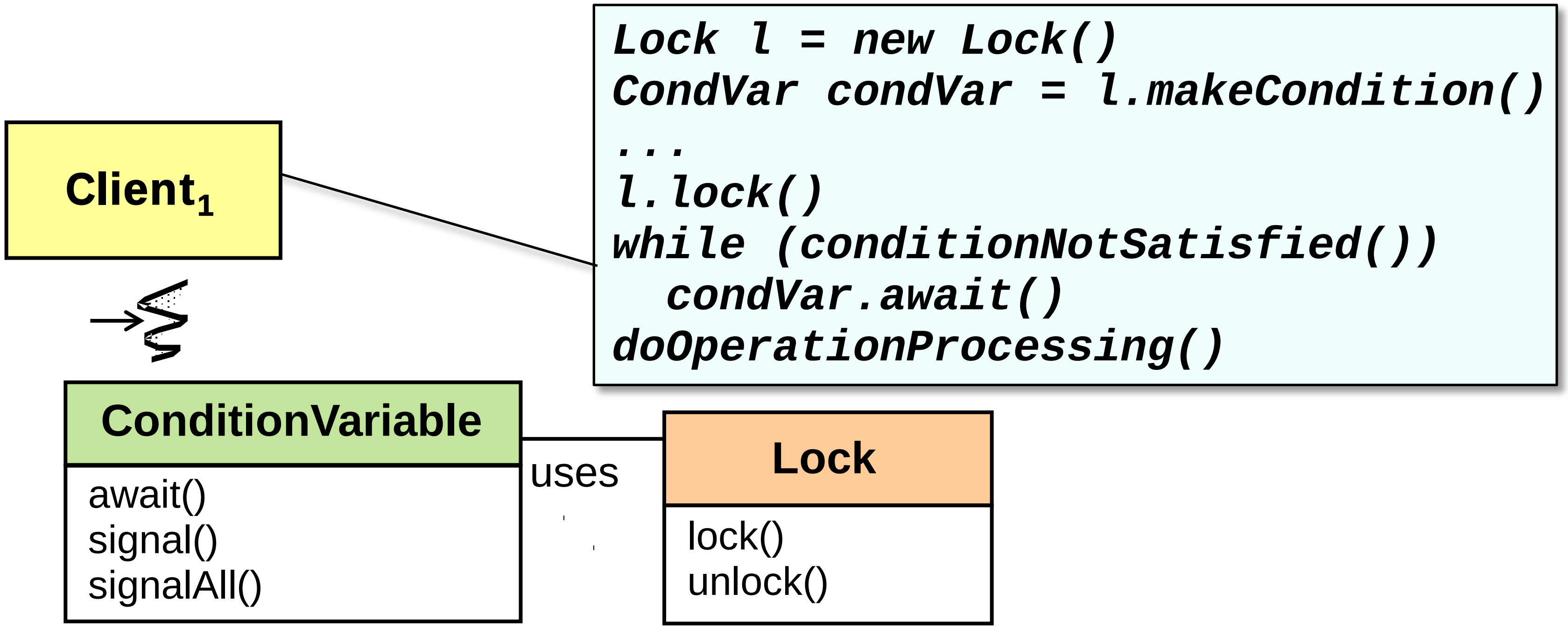
Overview of Condition Variables



```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true
  - A condition can be arbitrarily complex

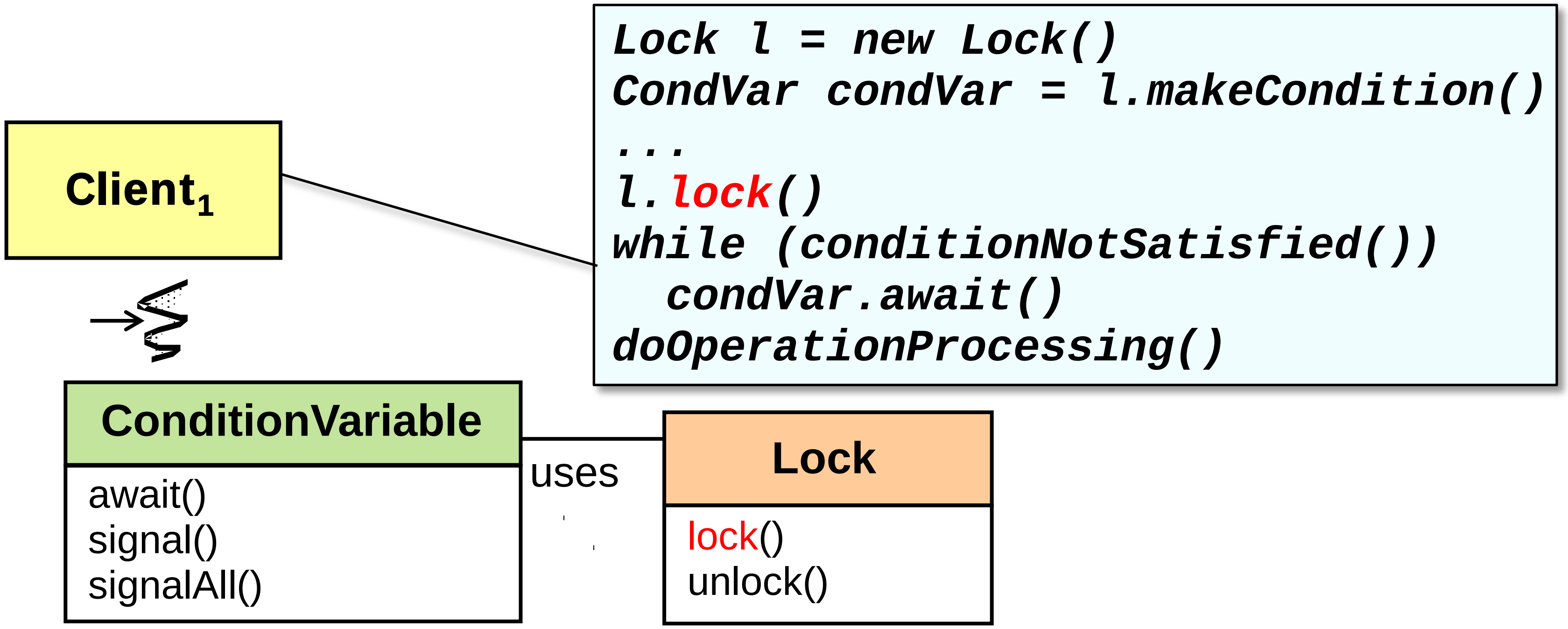
Overview of Condition Variables



```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true
  - A condition can be arbitrarily complex
    - A lock must therefore be associated with a condition variable to protect state shared between threads

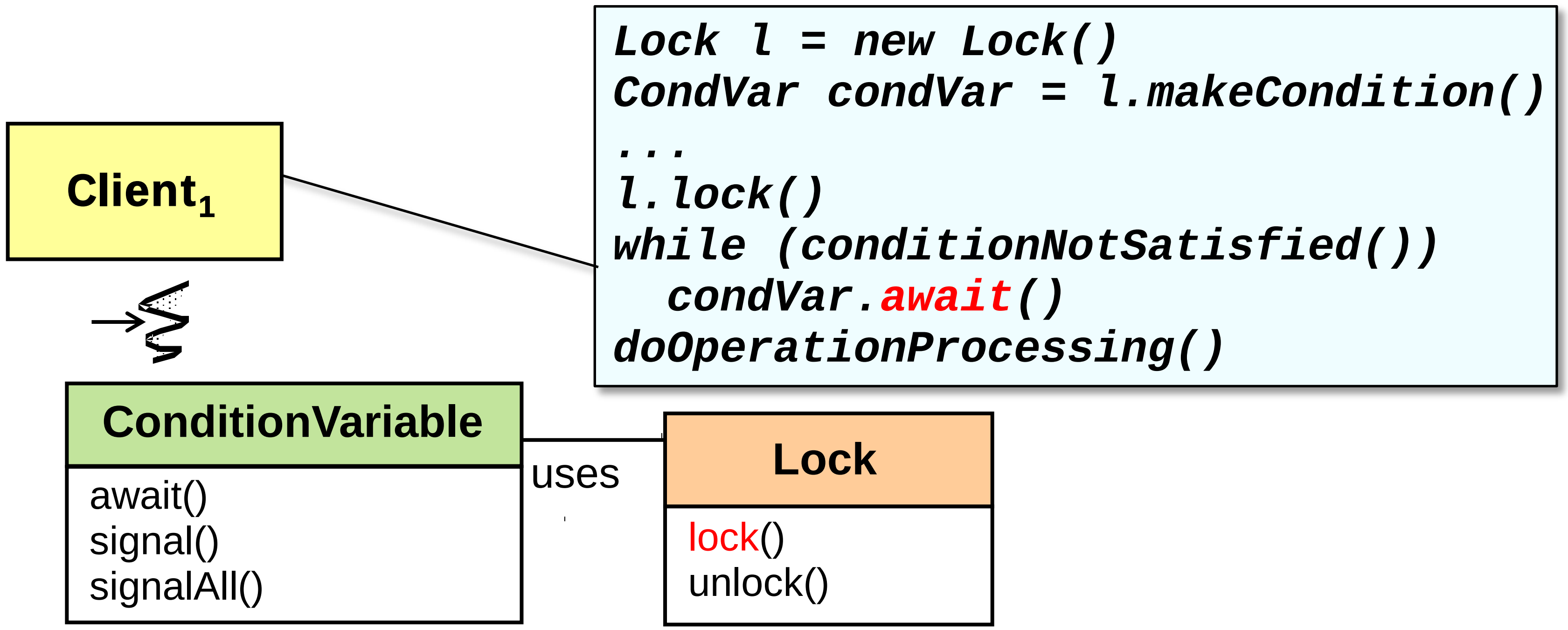
Overview of Condition Variables



```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true
  - A condition can be arbitrarily complex
    - A lock must therefore be associated with a condition variable to protect state shared between threads

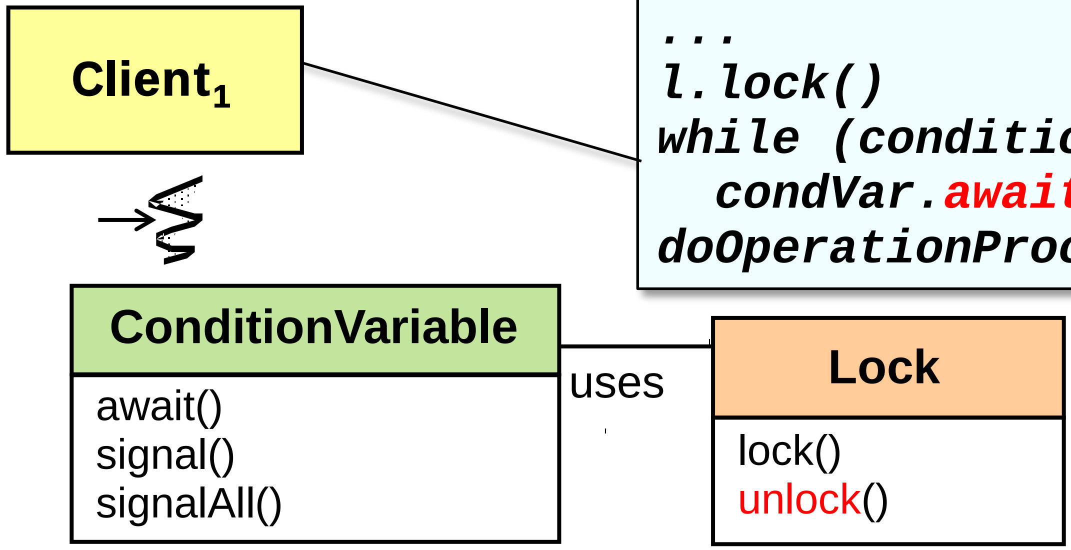
Overview of Condition Variables



```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true
  - A condition can be arbitrarily complex
  - Waiting on a condition variable releases lock & suspends thread atomically

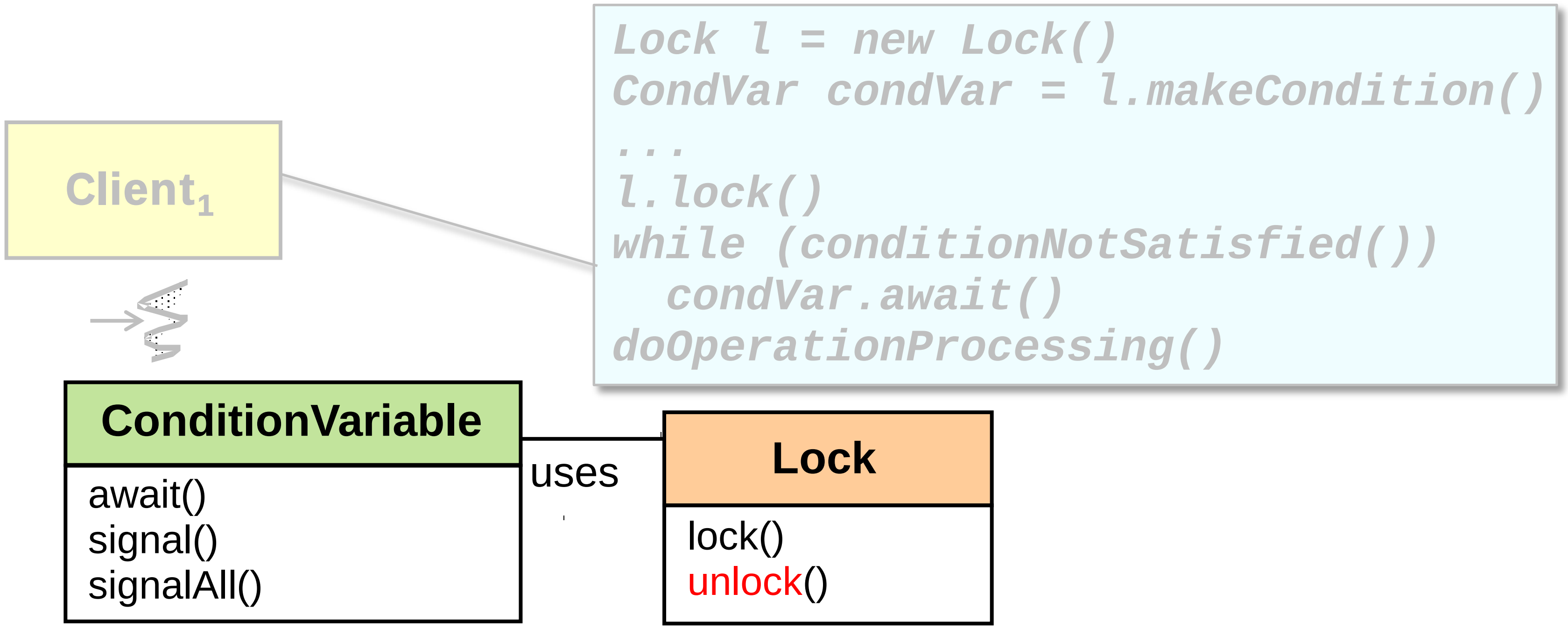
Overview of Condition Variables



```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true
  - A condition can be arbitrarily complex
  - Waiting on a condition variable releases lock & suspends thread atomically

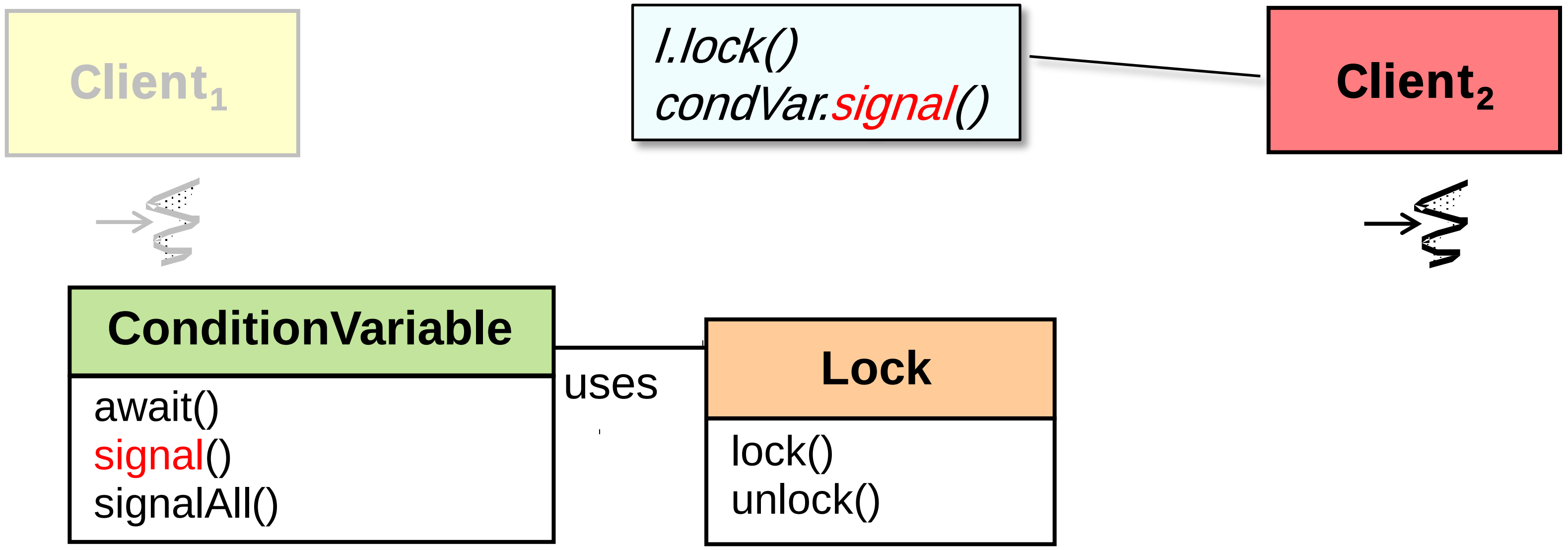
Overview of Condition Variables



```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

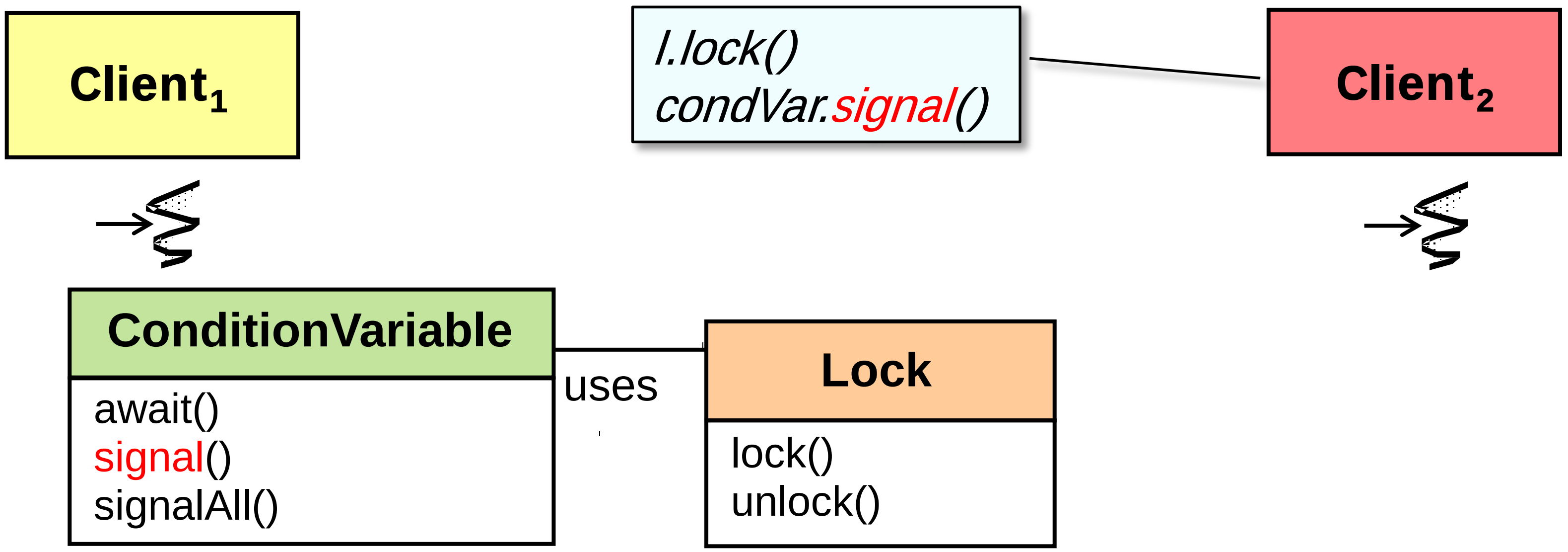
- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true
  - A condition can be arbitrarily complex
  - Waiting on a condition variable releases lock & suspends thread atomically

Overview of Condition Variables



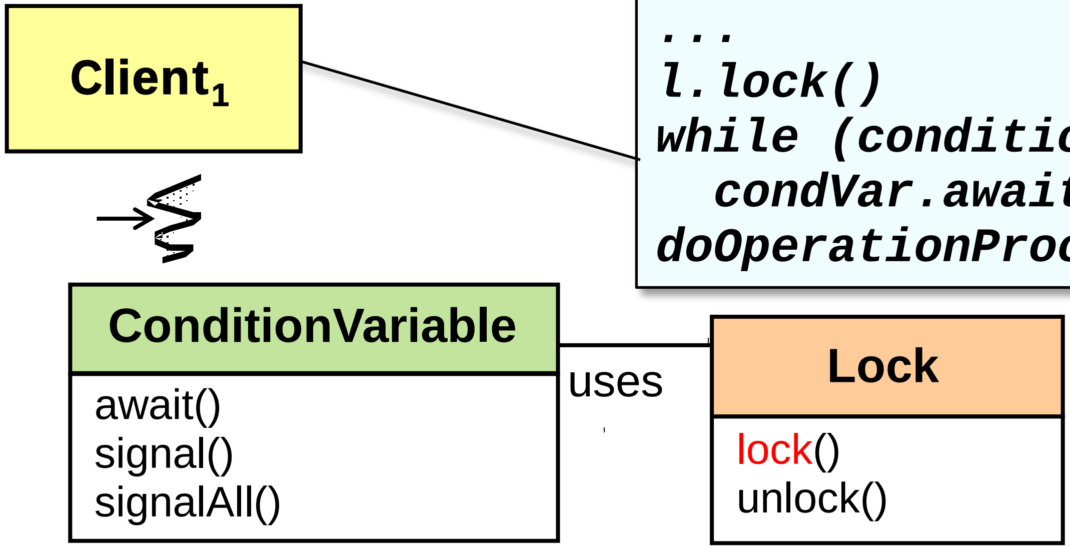
- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true
  - A condition can be arbitrarily complex
  - Waiting on a condition variable releases lock & suspends thread atomically
  - The thread suspended until another thread signals the condition variable

Overview of Condition Variables



- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true
  - A condition can be arbitrarily complex
  - Waiting on a condition variable releases lock & suspends thread atomically
  - The thread suspended until another thread signals the condition variable

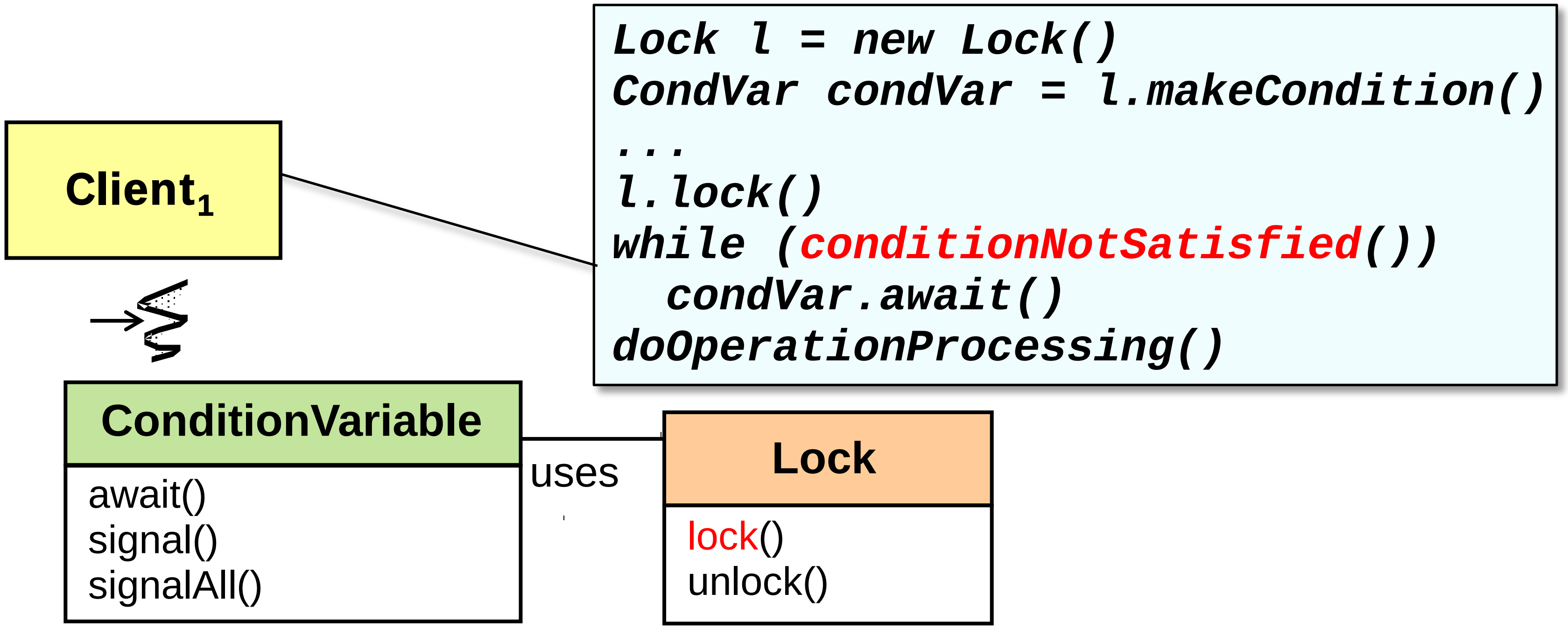
Overview of Condition Variables



```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true
  - A condition can be arbitrarily complex
  - Waiting on a condition variable releases lock & suspends thread atomically
  - The thread suspended until another thread signals the condition variable

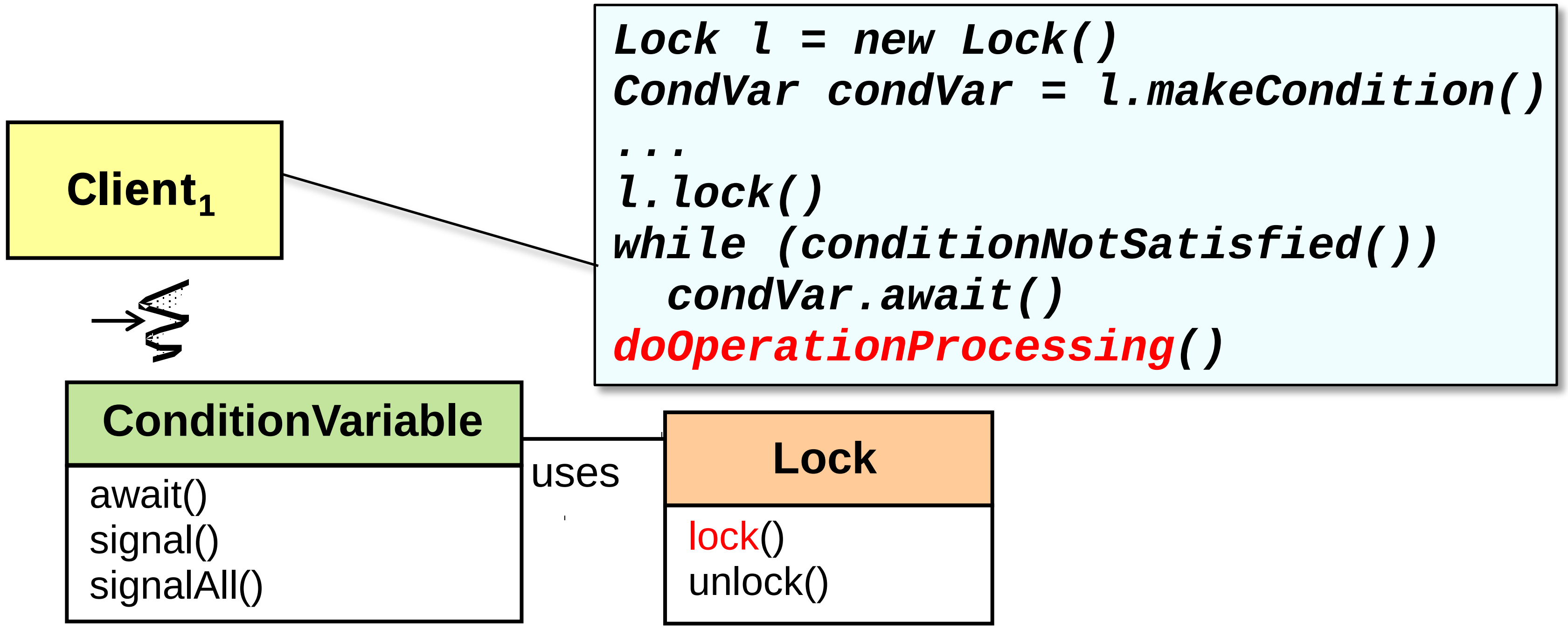
Overview of Condition Variables



```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true
  - A condition can be arbitrarily complex
  - Waiting on a condition variable releases lock & suspends thread atomically
  - The thread suspended until another thread signals the condition variable

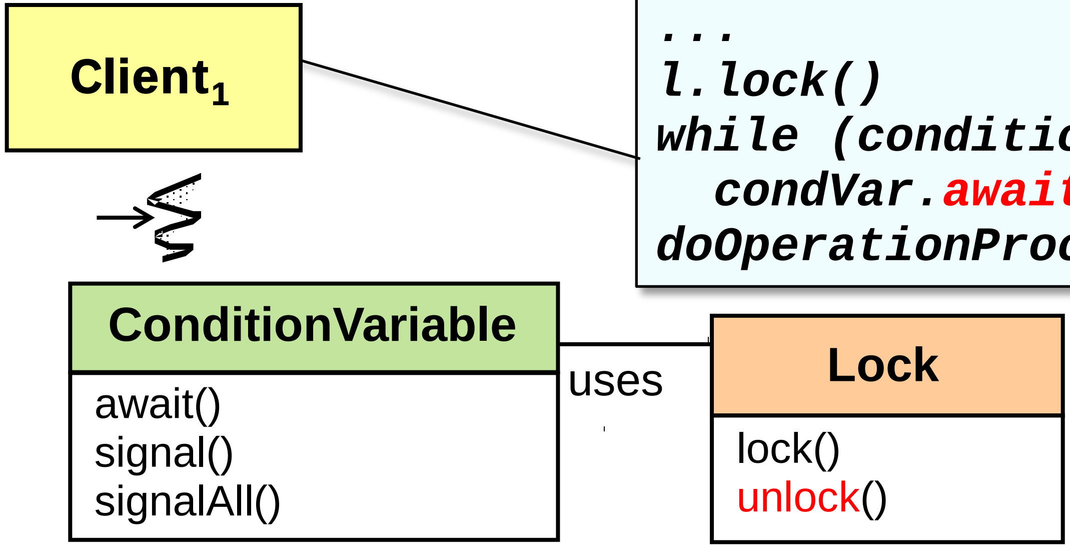
Overview of Condition Variables



```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true
  - A condition can be arbitrarily complex
  - Waiting on a condition variable releases lock & suspends thread atomically
  - The thread suspended until another thread signals the condition variable

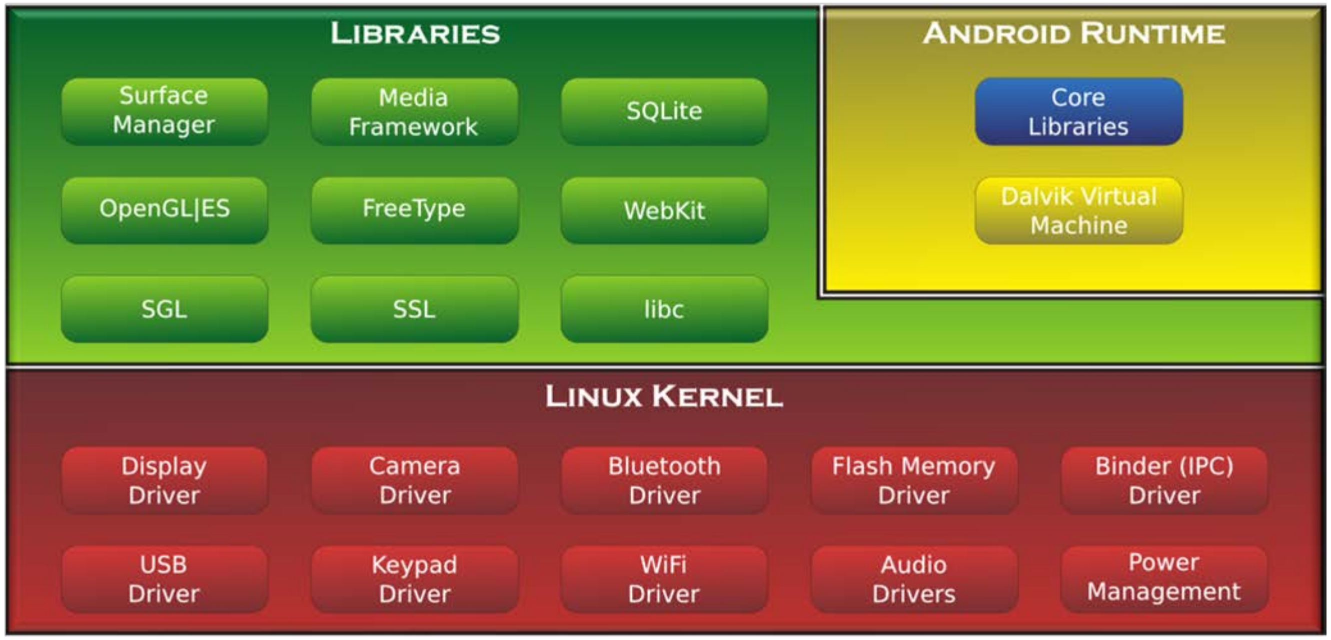
Overview of Condition Variables



```
Lock l = new Lock()
CondVar condVar = l.makeCondition()
...
l.lock()
while (conditionNotSatisfied())
  condVar.await()
doOperationProcessing()
```

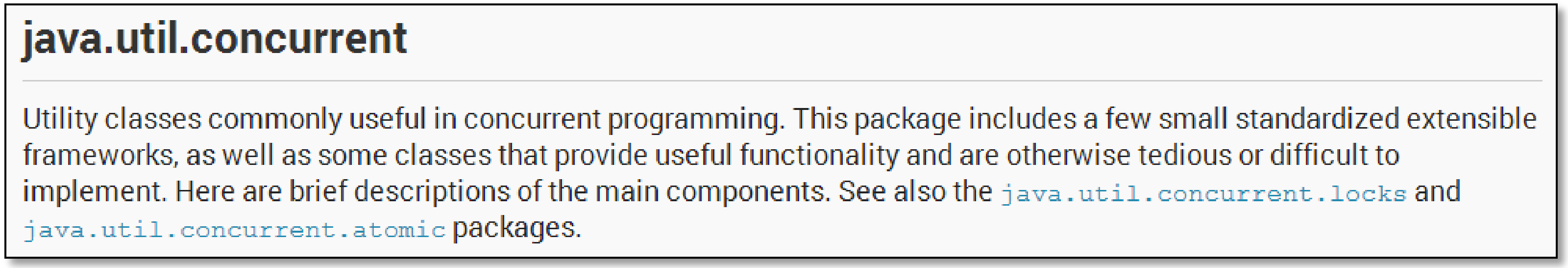
- A condition variable is used to implement the *Guarded Suspension* pattern
  - A thread can suspend its execution on a condition variable another thread notifies it that shared state it's waiting on may now be true
  - A condition can be arbitrarily complex
  - Waiting on a condition variable releases lock & suspends thread atomically
  - The thread suspended until another thread signals the condition variable

Overview of Condition Variables



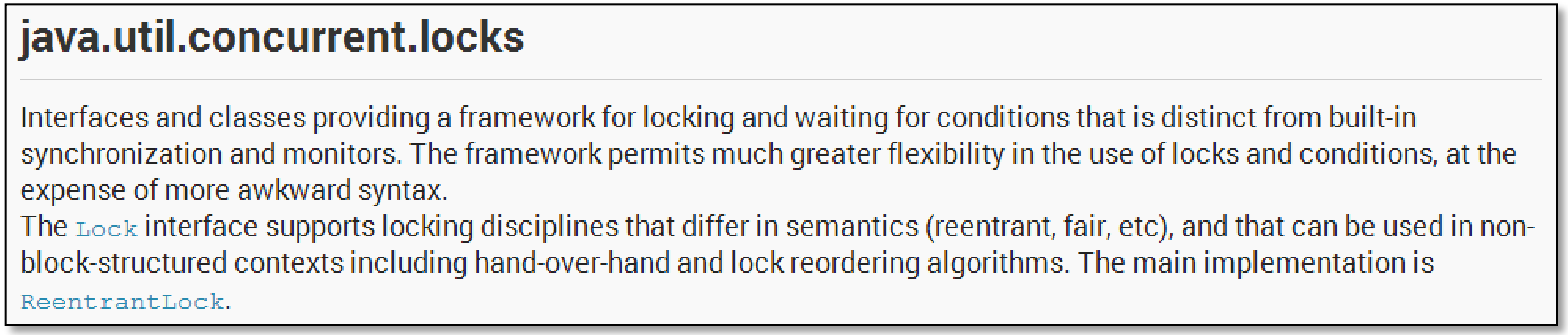
- A condition variable is used to implement the *Guarded Suspension* pattern
- Condition variables form the basis for synchronization & scheduling mechanisms in Java & Android

Overview of Condition Variables



java.util.concurrent

Utility classes commonly useful in concurrent programming. This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement. Here are brief descriptions of the main components. See also the `java.util.concurrent.locks` and `java.util.concurrent.atomic` packages.



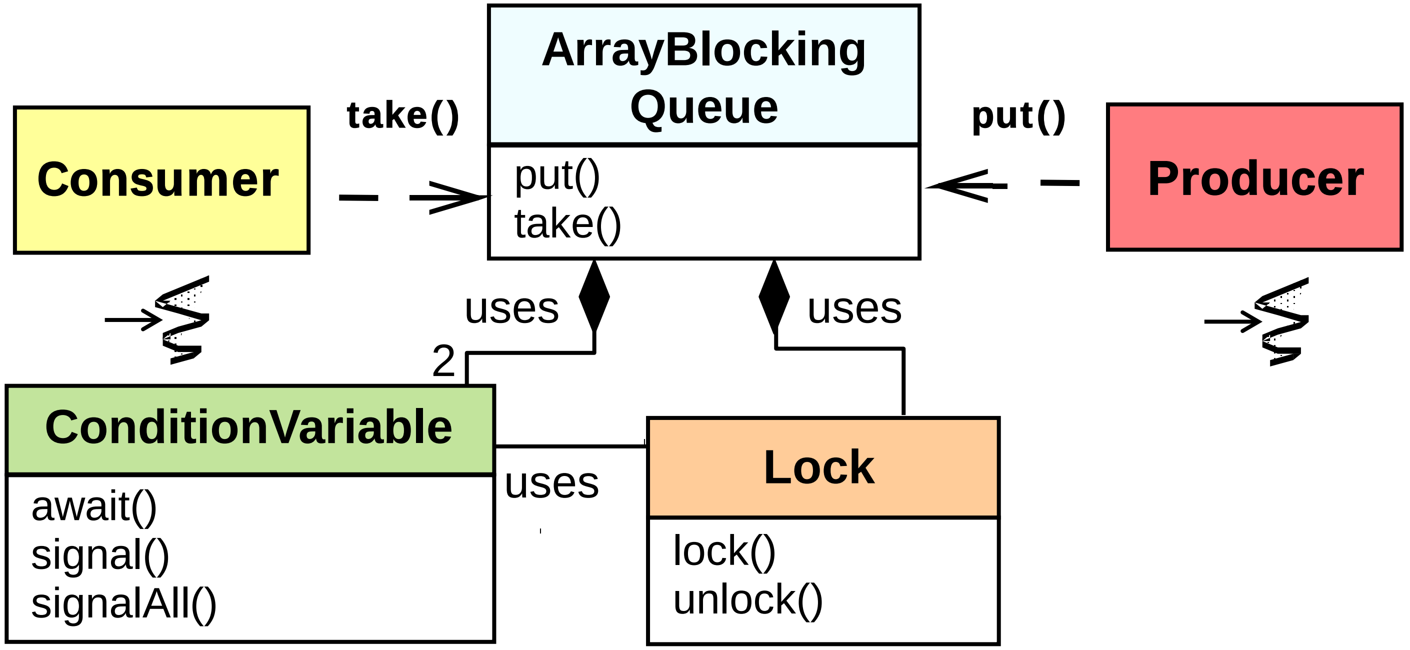
java.util.concurrent.locks

Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors. The framework permits much greater flexibility in the use of locks and conditions, at the expense of more awkward syntax.
The `Lock` interface supports locking disciplines that differ in semantics (reentrant, fair, etc), and that can be used in non-block-structured contexts including hand-over-hand and lock reordering algorithms. The main implementation is `ReentrantLock`.

- A condition variable is used to implement the *Guarded Suspension* pattern
- Condition variables form the basis for synchronization & scheduling mechanisms in Java & Android, e.g.
  - Blocking queues & deques in the java.util.concurrent* packages

developer.android.com/reference/java/util/concurrent/locks/AbstractQueuedLongSynchronizer.ConditionObject.html

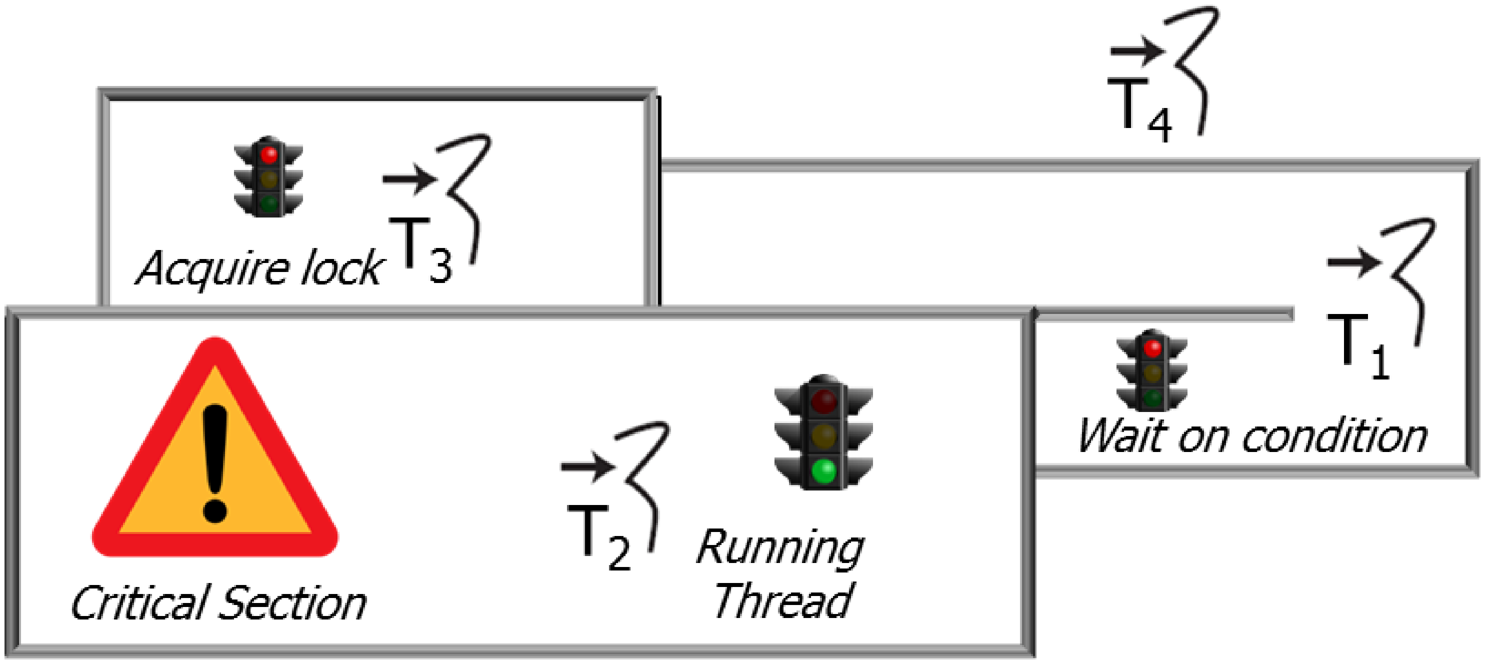
Overview of Condition Variables



- A condition variable is used to implement the *Guarded Suspension* pattern
- Condition variables form the basis for synchronization & scheduling mechanisms in Java & Android, e.g.
  - Blocking queues & deques in the java.util.concurrent* packages

developer.android.com/reference/java/util/concurrent/ArrayBlockingQueue.html

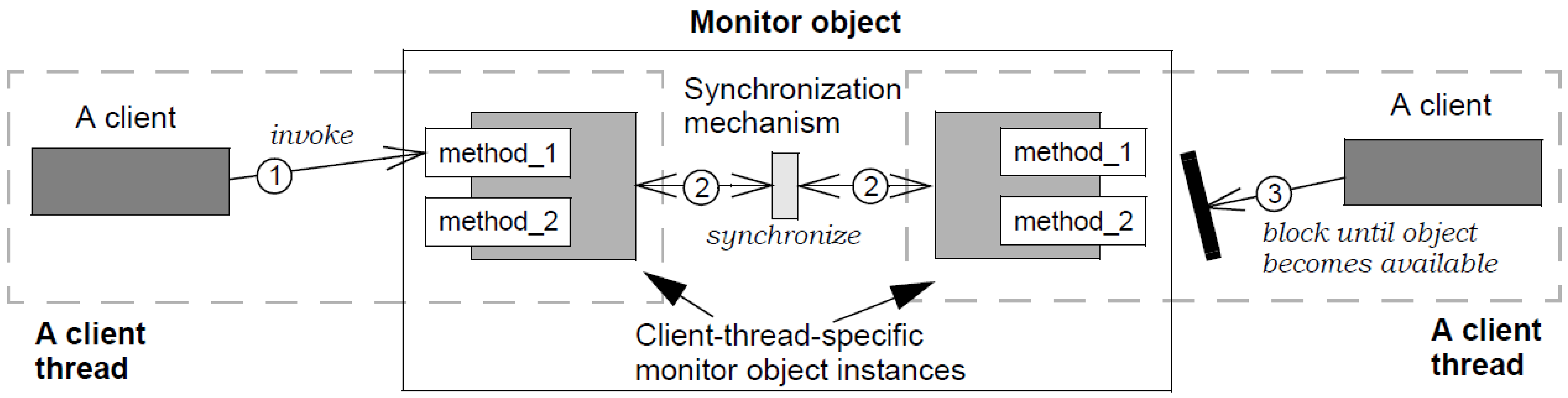
Overview of Condition Variables



- A condition variable is used to implement the *Guarded Suspension* pattern
- Condition variables form the basis for synchronization & scheduling mechanisms in Java & Android, e.g.
  - Blocking queues & deques in the java.util.concurrent* packages
  - Java built-in monitor objects

See upcoming part on Java Built-in Monitor Objects

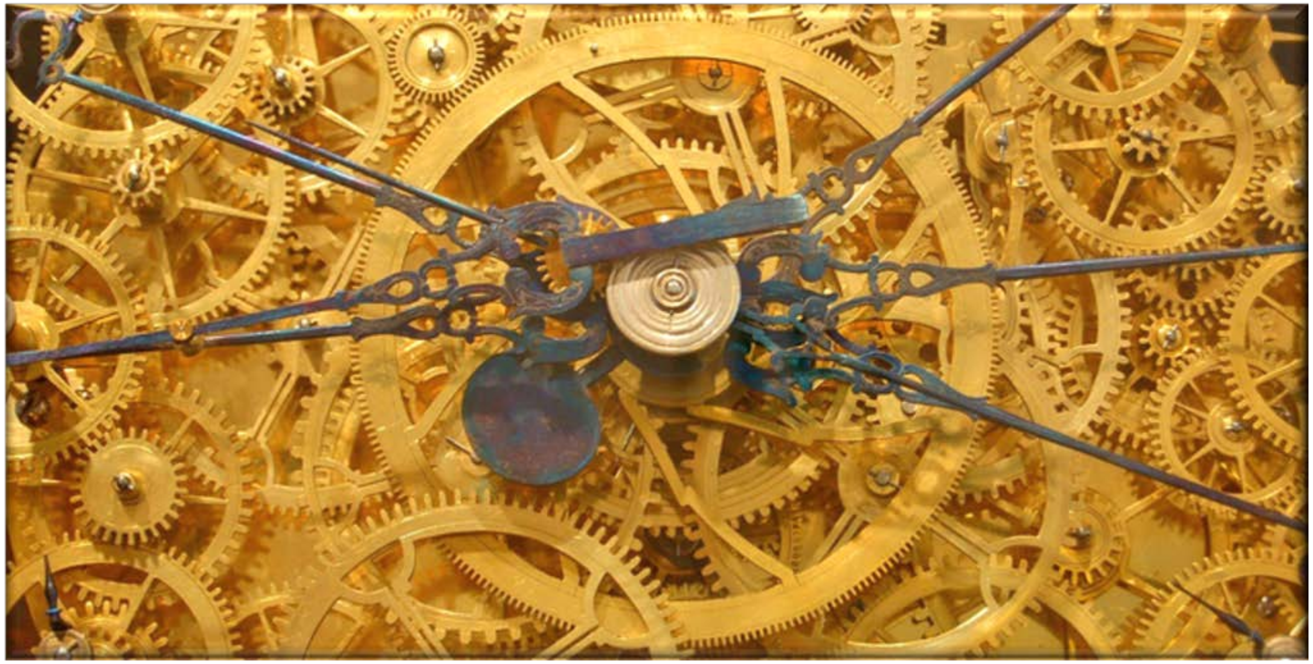
Overview of Condition Variables



- A condition variable is used to implement the *Guarded Suspension* pattern
- Condition variables form the basis for synchronization & scheduling mechanisms in Java & Android, e.g.
  - Blocking queues & deques in the java.util.concurrent* packages
  - Java built-in monitor objects
  - The *Monitor Object* pattern

See upcoming parts on the *Monitor Object* pattern

Overview of Condition Variables



- A condition variable is used to implement the *Guarded Suspension* pattern
- Condition variables form the basis for synchronization & scheduling mechanisms in Java & Android
- Condition variables are powerful, but can be hard to understand & apply

You might want to rewatch this video & read the links carefully

Human Known Use of Condition Variables

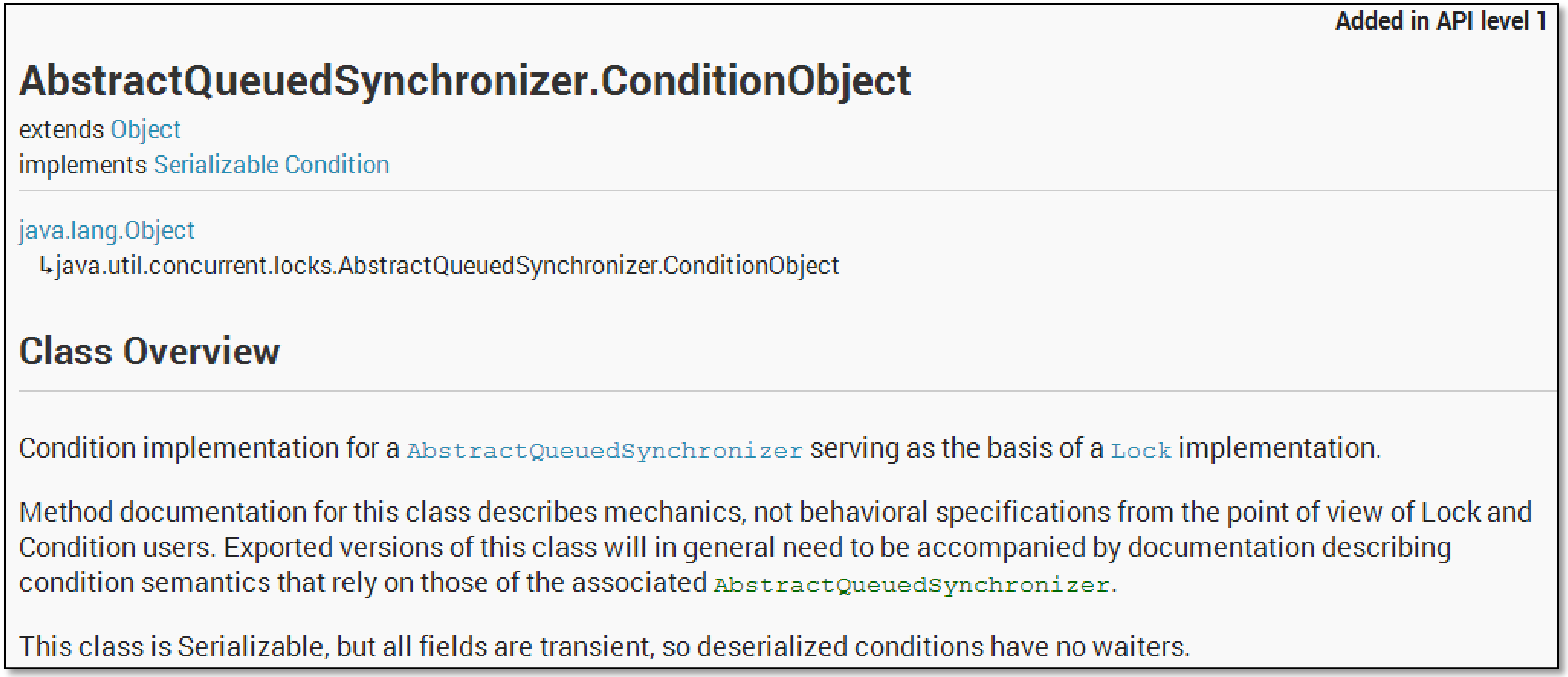
Overview of Condition Variables

- A condition variable is used to implement the *Guarded Suspension* pattern
- Condition variables form the basis for synchronization & scheduling mechanisms in Java & Android
- Condition variables are powerful, but can be hard to understand & apply
- A human known use is a pizza delivery protocol

Overview of ConditionObject

Overview of ConditionObject

- ConditionObject implements the Condition interface & is defined in AbstractQueuedSynchronizer



Added in API level 1

AbstractQueuedSynchronizer.ConditionObject

extends Object
implements Serializable Condition

java.lang.Object
  ↳java.util.concurrent.locks.AbstractQueuedSynchronizer.ConditionObject

Class Overview

Condition implementation for a `AbstractQueuedSynchronizer` serving as the basis of a `Lock` implementation.

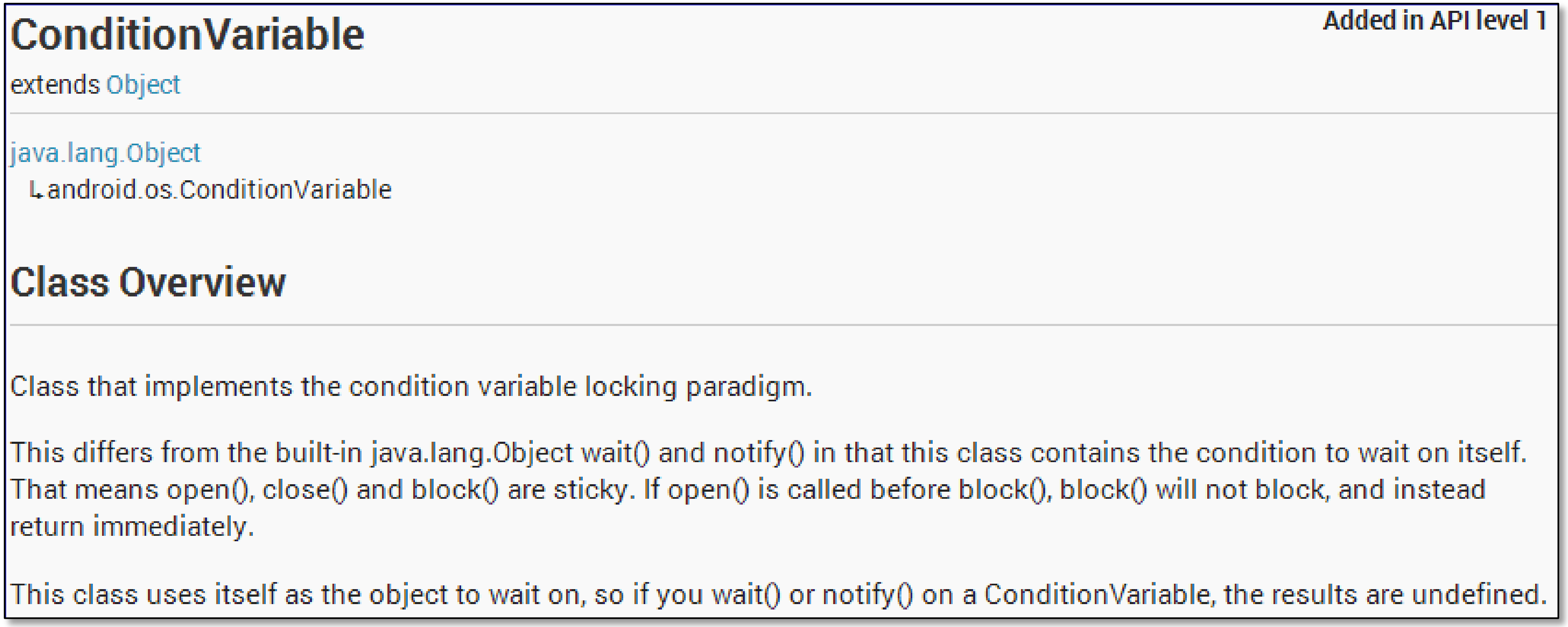
Method documentation for this class describes mechanics, not behavioral specifications from the point of view of Lock and Condition users. Exported versions of this class will in general need to be accompanied by documentation describing condition semantics that rely on those of the associated `AbstractQueuedSynchronizer`.

This class is Serializable, but all fields are transient, so deserialized conditions have no waiters.

developer.android.com/reference/java/util/concurrent/locks/AbstractQueuedSynchronizer.ConditionObject.html

Overview of ConditionObject

- ConditionObject implements the Condition interface & is defined in AbstractQueuedSynchronizer
  - Android also implements ConditionVariable



Added in API level 1

ConditionVariable

extends Object

java.lang.Object
↳android.os.ConditionVariable

Class Overview

Class that implements the condition variable locking paradigm.

This differs from the built-in java.lang.Object wait() and notify() in that this class contains the condition to wait on itself. That means open(), close() and block() are sticky. If open() is called before block(), block() will not block, and instead return immediately.

This class uses itself as the object to wait on, so if you wait() or notify() on a ConditionVariable, the results are undefined.

developer.android.com/reference/android/os/ConditionVariable.html has more

Overview of ConditionObject

- ConditionObject implements the Condition interface & is defined in AbstractQueuedSynchronizer
  - Android also implements ConditionVariable
  - It is mostly written in Java

```
public class ConditionObject
             implements Condition,
             java.io.Serializable {
  ...
...
```

libcore/luni/src/main/java/java/util/concurrent/locks/AbstractQueuedSynchronizer.java

Overview of ConditionObject

- ConditionObject implements the Condition interface & is defined in AbstractQueuedSynchronizer
- Its key methods are await(), signal(), & signalAll()

```
public class ConditionObject
             implements Condition,
             java.io.Serializable {
  ...
  /** Implement interruptible
      condition wait. */
  public final void await()
    throws InterruptedException
  { ... }

  /** Wakeup the longest waiting
      thread. */
  public final void signal()
  { ... }

  /** Wakeup all waiting threads. */
  public final void signalAll()
  { ...}
  ...
```

Overview of ConditionObject

- ConditionObject implements the Condition interface & is defined in AbstractQueuedSynchronizer
- Its key methods are await(), signal(), & signalAll()
  - Similar to Java's built-in monitor object methods

```
public class ConditionObject
                implements Condition,
                java.io.Serializable {
  ...
  /** Implement interruptible
      condition wait. */
  public final void await()
    throws InterruptedException
  { ... }

  /** Wakeup the longest waiting
      thread. */
  public final void signal()
  { ... }

  /** Wakeup all waiting threads. */
  public final void signalAll()
  { ...}
  ...
```

See upcoming part on "Java Built-in Monitor Objects"

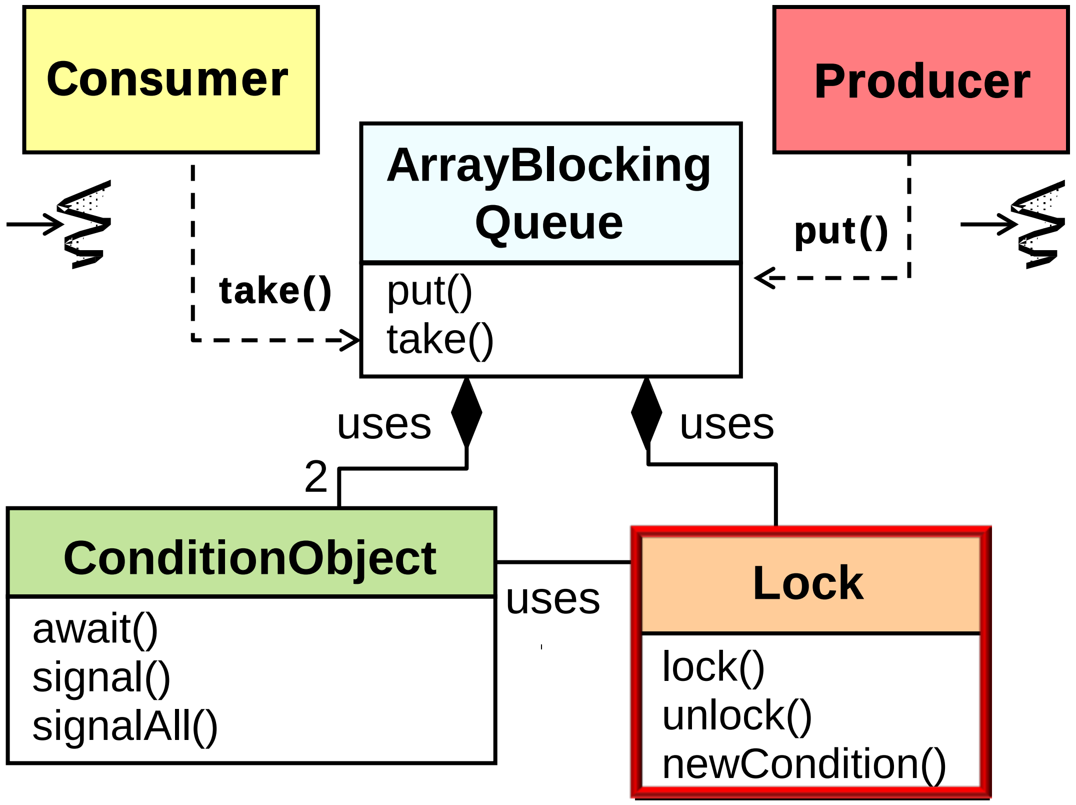
Overview of ConditionObject

- ConditionObject implements the Condition interface & is defined in AbstractQueuedSynchronizer
- Its key methods are await(), signal(), & signalAll()
  - Similar to Java's built-in monitor object methods
  - Internally, several queues are used to enable threads to schedule their interactions

```
public class ConditionObject
             implements Condition,
             java.io.Serializable {
  ...
  /** First node of condition queue.
    */
  private transient Node
    firstWaiter;
   /** Last node of condition queue.
     */
   private transient Node
    lastWaiter;
  ...
```

Overview of ConditionObject

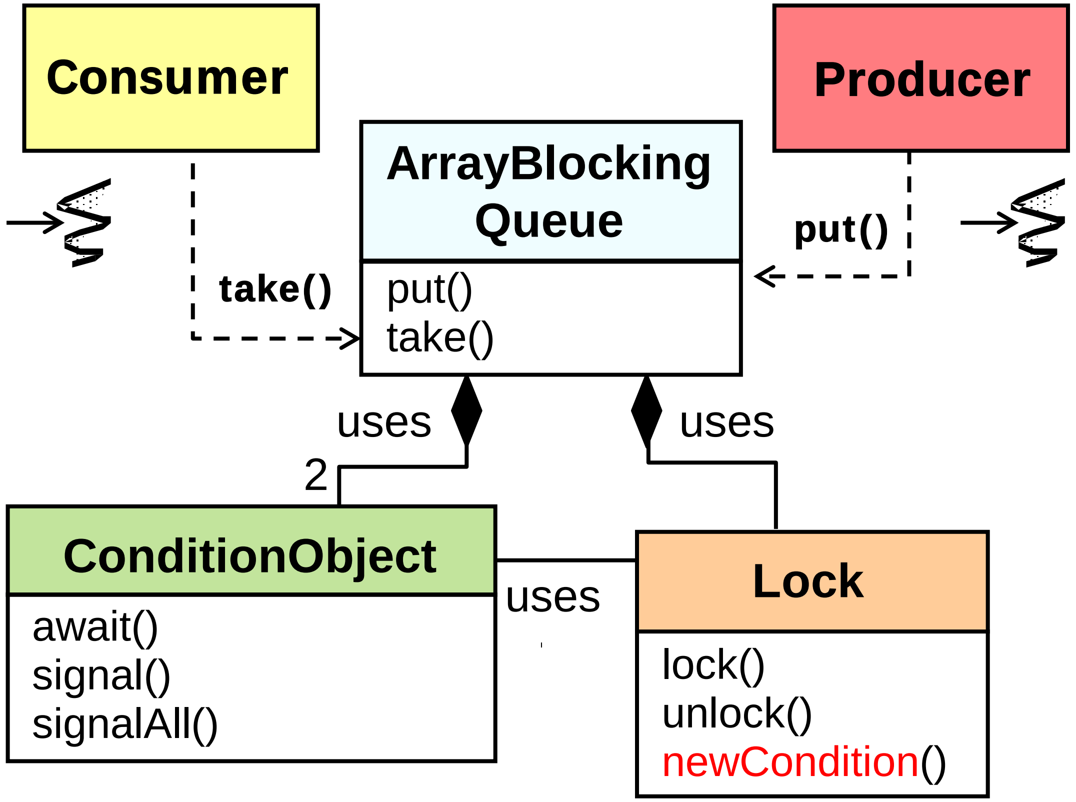
- ConditionObject implements the Condition interface & is defined in AbstractQueuedSynchronizer
- Its key methods are await(), signal(), & signalAll()
- A ConditionObject is always used in conjunction with a Lock



See earlier part on “Java ReentrantLock”

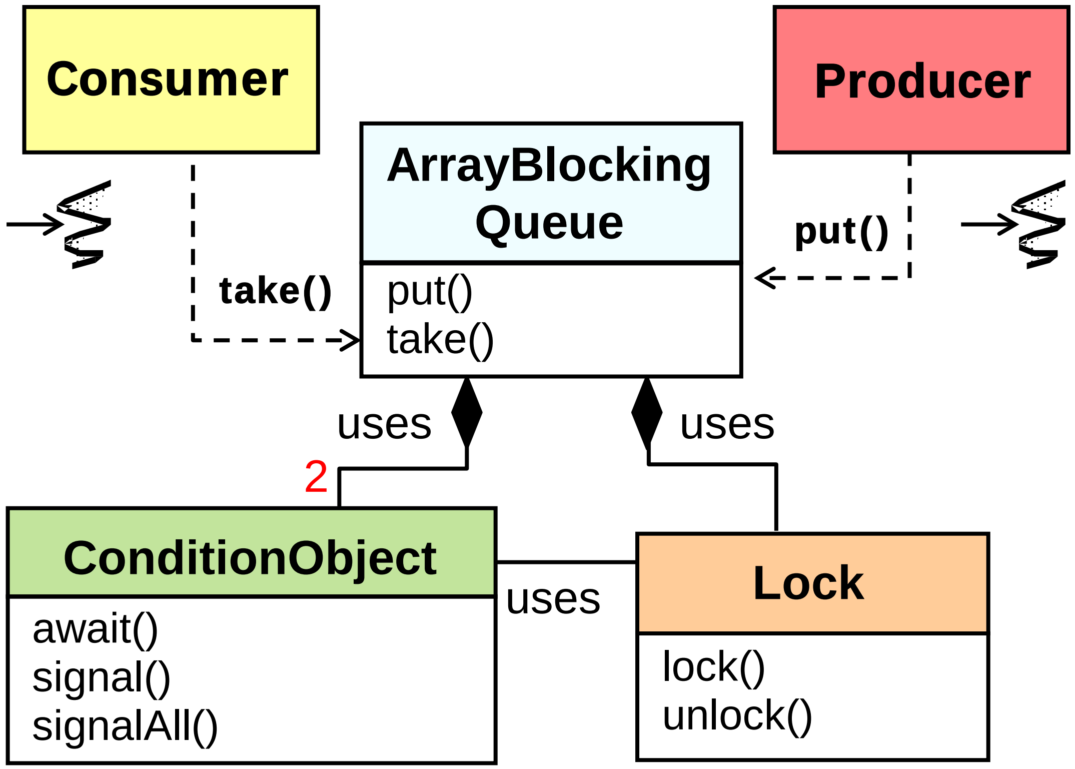
Overview of ConditionObject

- ConditionObject implements the Condition interface & is defined in AbstractQueuedSynchronizer
- Its key methods are await(), signal(), & signalAll()
- A ConditionObject is always used in conjunction with a Lock
  - The newCondition() method on ReentrantLock returns a ConditionObject that can be used with this lock



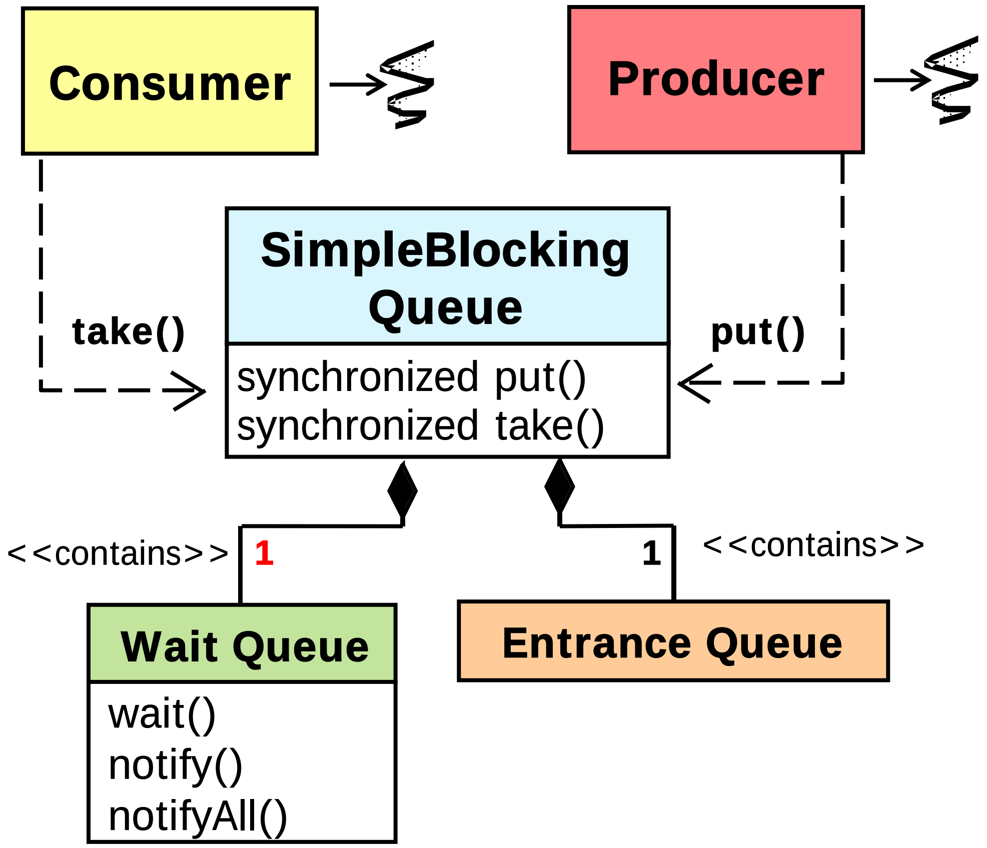
Overview of ConditionObject

- ConditionObject implements the Condition interface & is defined in AbstractQueuedSynchronizer
- Its key methods are await(), signal(), & signalAll()
- A ConditionObject is always used in conjunction with a Lock
- User-defined objects can have multiple ConditionObjects



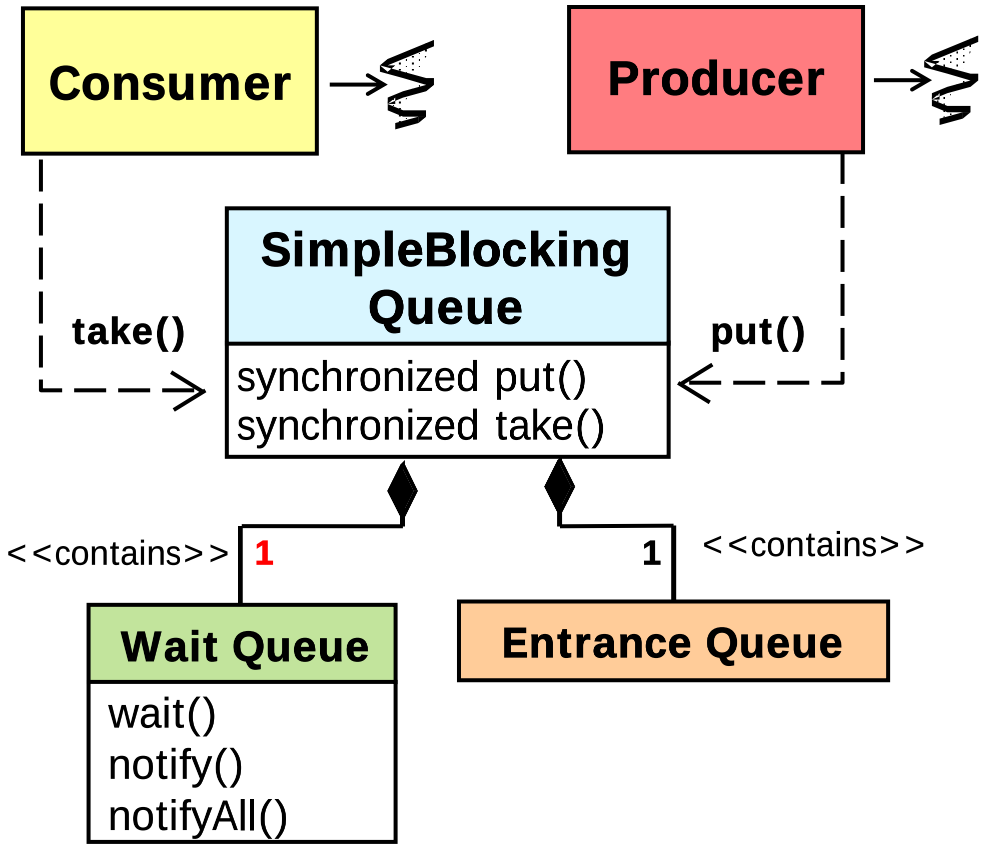
Overview of ConditionObject

- ConditionObject implements the Condition interface & is defined in AbstractQueuedSynchronizer
- Its key methods are await(), signal(), & signalAll()
- A ConditionObject is always used in conjunction with a Lock
- User-defined objects can have multiple ConditionObjects
    - Java's built-in monitor objects only support a single monitor condition (known as its "wait queue")



Overview of ConditionObject

- ConditionObject implements the Condition interface & is defined in AbstractQueuedSynchronizer
- Its key methods are await(), signal(), & signalAll()
- A ConditionObject is always used in conjunction with a Lock
- User-defined objects can have multiple ConditionObjects
  - Java's built-in monitor objects only support a single monitor condition (known as its "wait queue")

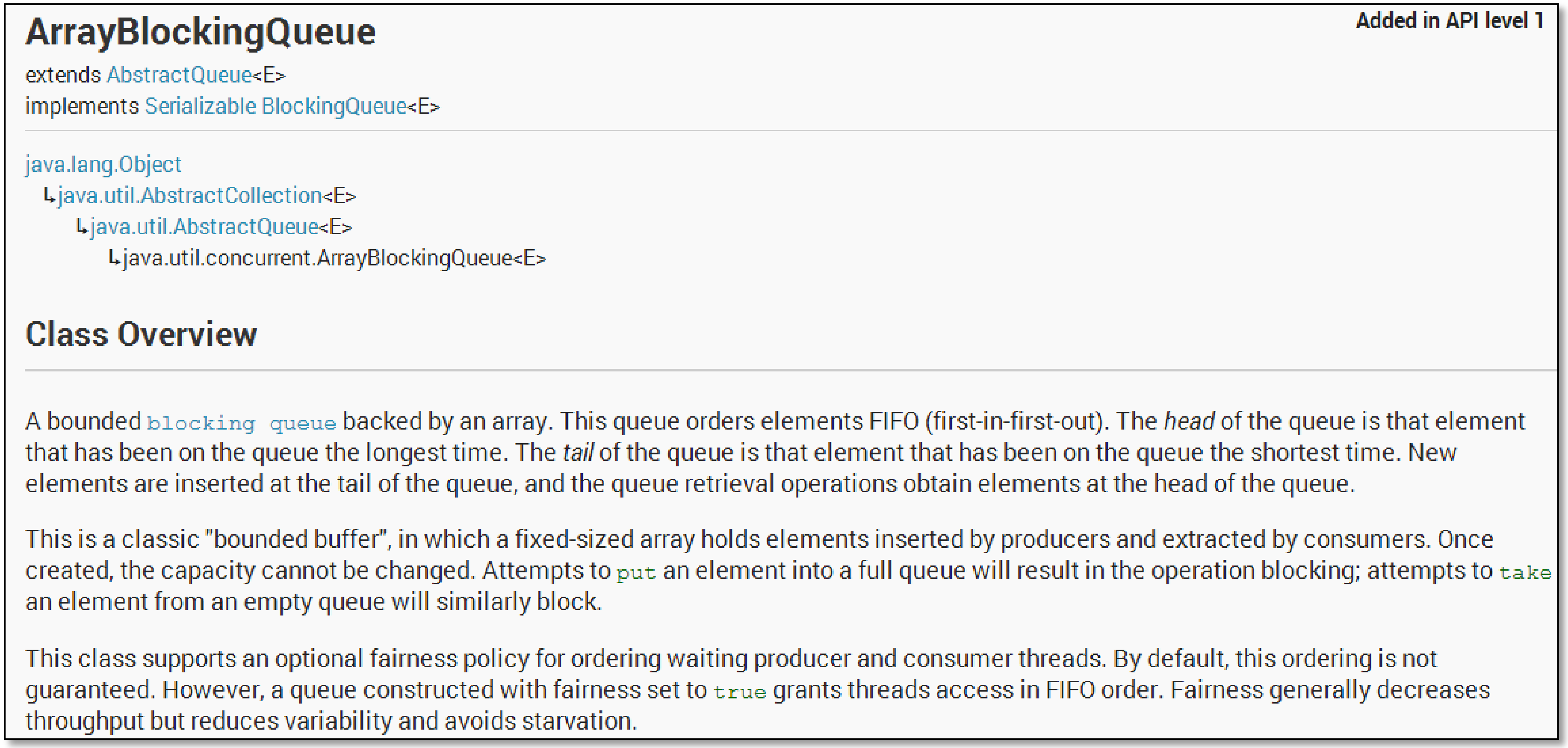


See upcoming part on "Java Built-in Monitor Objects"

Using ConditionObject in Android

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue



ArrayBlockingQueue

Added in API level 1

extends AbstractQueue<E>
implements Serializable BlockingQueue<E>

java.lang.Object
 ↳java.util.AbstractCollection<E>
  ↳java.util.AbstractQueue<E>
   ↳java.util.concurrent.ArrayBlockingQueue<E>

Class Overview

A bounded `blocking queue` backed by an array. This queue orders elements FIFO (first-in-first-out). The *head* of the queue is that element that has been on the queue the longest time. The *tail* of the queue is that element that has been on the queue the shortest time. New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

This is a classic "bounded buffer", in which a fixed-sized array holds elements inserted by producers and extracted by consumers. Once created, the capacity cannot be changed. Attempts to `put` an element into a full queue will result in the operation blocking; attempts to `take` an element from an empty queue will similarly block.

This class supports an optional fairness policy for ordering waiting producer and consumer threads. By default, this ordering is not guaranteed. However, a queue constructed with fairness set to `true` grants threads access in FIFO order. Fairness generally decreases throughput but reduces variability and avoids starvation.

developer.android.com/reference/java/util/concurrent/ArrayBlockingQueue.html

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue

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

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue

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

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue

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

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue

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

See libcore/luni/src/main/java/java/util/concurrent/ArrayBlockingQueue.java

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue

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

See earlier part on “Java ReentrantLock”

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - It's implemented using an dynamically sized array

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  /** The queued items */
  final Object[] items;

  /** items index for next take,
       poll, peek or remove */
  int takeIndex;

  /** items index for next put,
      offer, or add */
  int putIndex;
  ...
```

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - It’s implemented using an dynamically sized array

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  /** The queued items */
  final Object[] items;

  /** items index for next take,
       poll, peek or remove */
  int takeIndex;

  /** items index for next put,
      offer, or add */
  int putIndex;
  ...
```

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - It's implemented using an dynamically sized array
  - It has a ReentrantLock & two ConditionObjects

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  /** Main lock guarding access */
  final ReentrantLock lock;

  /** Condition for waiting takes */
  private final Condition notEmpty;

  /** Condition for waiting puts */
  private final Condition notFull;
  ...
```

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - It's implemented using an dynamically sized array
  - It has a ReentrantLock & two ConditionObjects

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  /** Main lock guarding access */
  final ReentrantLock lock;

  /** Condition for waiting takes */
  private final Condition notEmpty;

  /** Condition for waiting puts */
  private final Condition notFull;
  ...
```

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - It's implemented using an dynamically sized array
  - It has a ReentrantLock & two ConditionObjects

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  /** Main lock guarding access */
  final ReentrantLock lock;

  /** Condition for waiting takes */
  private final Condition notEmpty;

  /** Condition for waiting puts */
  private final Condition notFull;
  ...
```

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - It's implemented using an dynamically sized array
  - It has a ReentrantLock & two ConditionObjects

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  public ArrayBlockingQueue
          (int capacity,
           boolean fair) {
    items =
      new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull =  lock.newCondition();
  }
```

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
  - It's implemented using an dynamically sized array
  - It has a ReentrantLock & two ConditionObjects

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  public ArrayBlockingQueue
          (int capacity,
           boolean fair) {
    items =
      new Object[capacity];
    lock = new ReentrantLock(fair);
    notEmpty = lock.newCondition();
    notFull =  lock.newCondition();
  }
```

Using ConditionObject in Android (continued)

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  /** Main lock guarding access */
  final ReentrantLock lock;

  /** Condition for waiting takes */
  private final Condition notEmpty;

  /** Condition for waiting puts */
  private final Condition notFull;
  ...

}
```

www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf presents *Monitor Object*

Using ConditionObject in Android

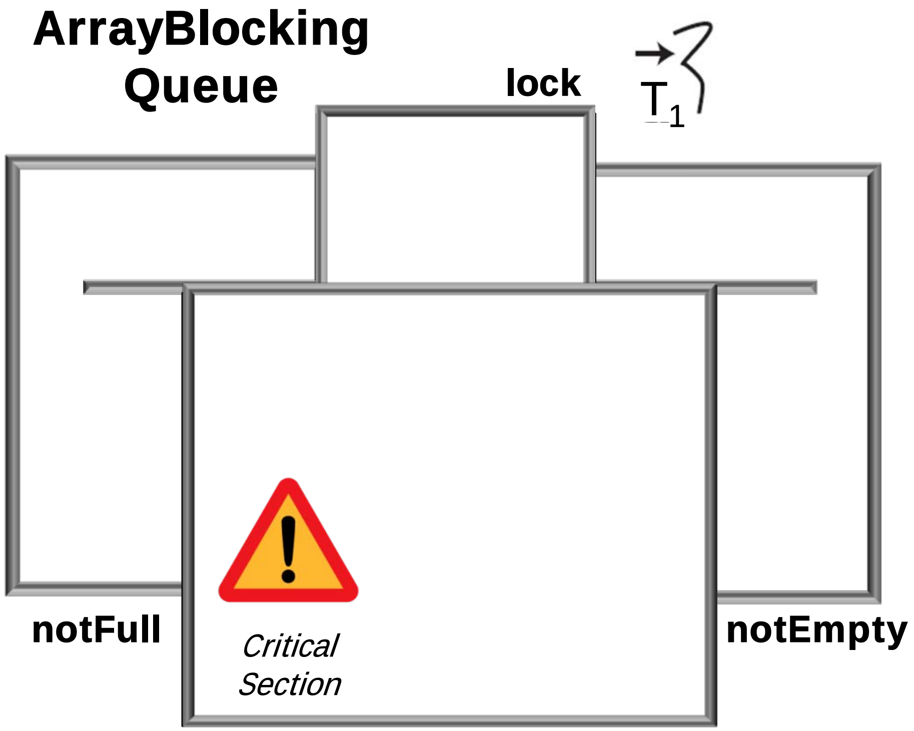
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  /** Main lock guarding access */
  final ReentrantLock lock;

  /** Condition for waiting takes */
  private final Condition notEmpty;

  /** Condition for waiting puts */
  private final Condition notFull;
  ...

}
```

See upcoming parts on “The Monitor Object Pattern”

Using ConditionObject in Android

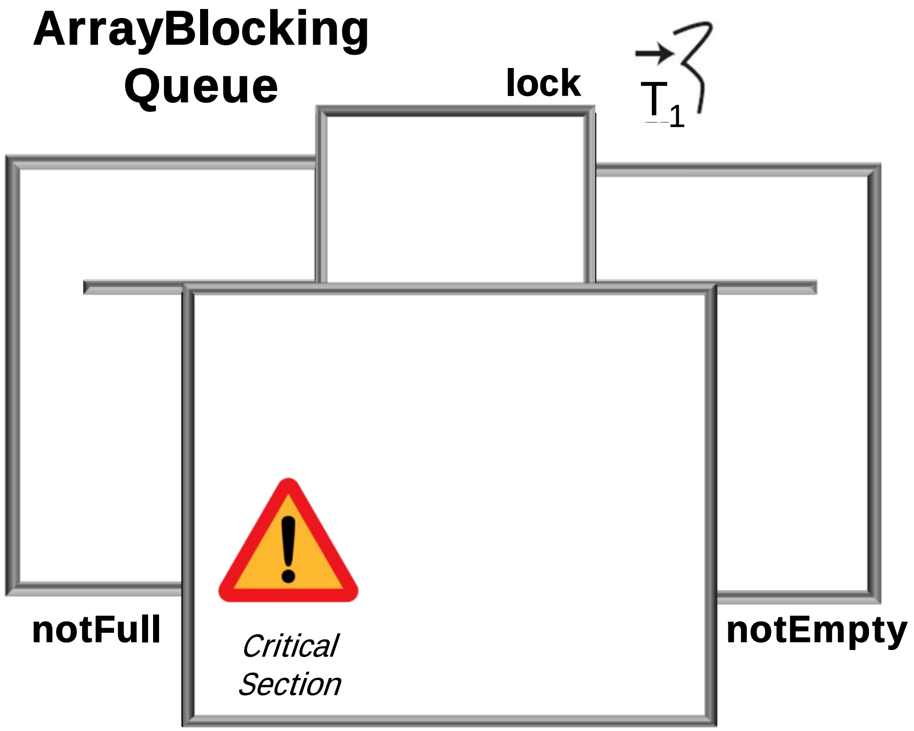
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern



```
public class ArrayBlockingQueue<E>
                   extends AbstractQueue<E>
                implements BlockingQueue<E>,
                      java.io.Serializable {
  ...
  /** Main lock guarding access */
  final ReentrantLock lock;

  /** Condition for waiting takes */
  private final Condition notEmpty;

  /** Condition for waiting puts */
  private final Condition notFull;
  ...
```

Using ConditionObject in Android

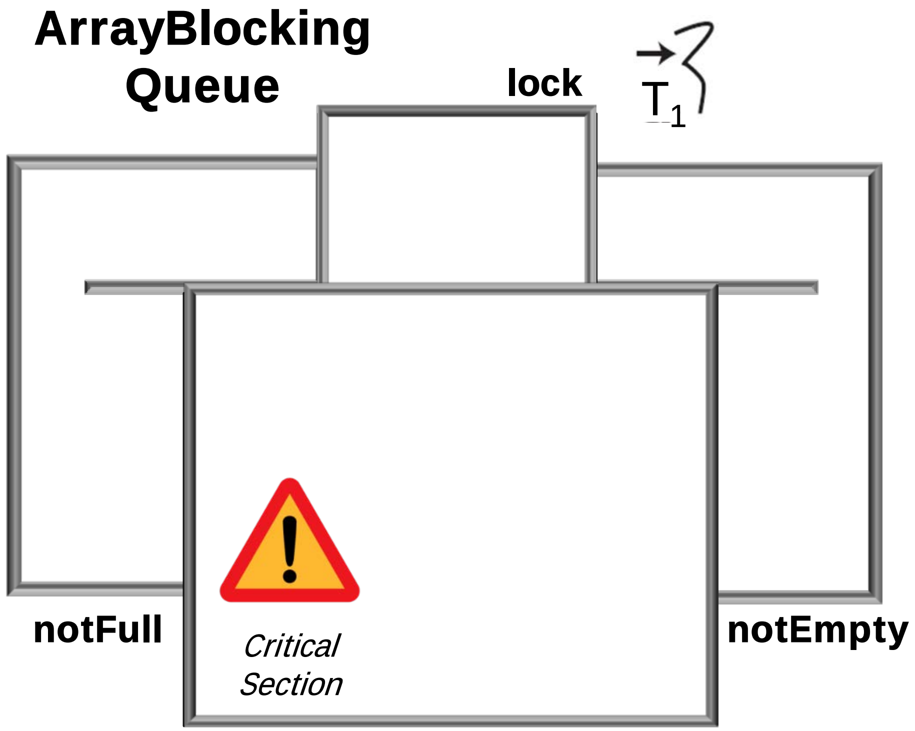
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern



```
public class ArrayBlockingQueue<E>
                   extends AbstractQueue<E>
                implements BlockingQueue<E>,
                      java.io.Serializable {
  ...
  /** Main lock guarding access */
  final ReentrantLock lock;

  /** Condition for waiting takes */
  private final Condition notEmpty;

  /** Condition for waiting puts */
  private final Condition notFull;
  ...
```

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern



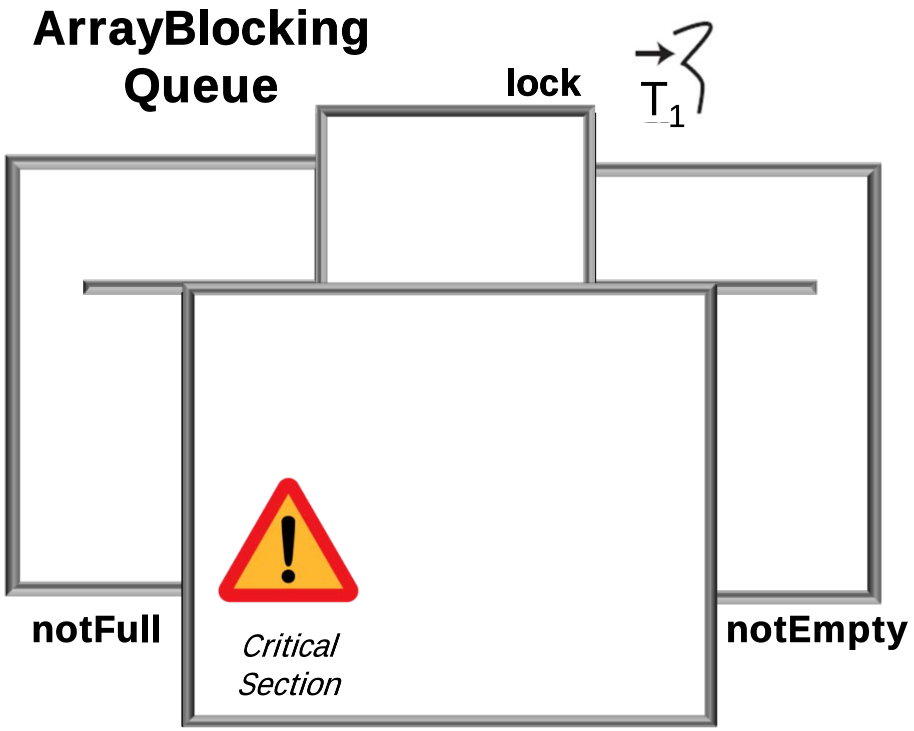
```
public class ArrayBlockingQueue<E>
            extends AbstractQueue<E>
         implements BlockingQueue<E>,
               java.io.Serializable {
  ...
  /** Main lock guarding access */
  final ReentrantLock lock;

  /** Condition for waiting takes */
  private final Condition notEmpty;

  /** Condition for waiting puts */
  private final Condition notFull;
  ...
```

These steps apply to the *Monitor Object* pattern & ConditionObjects in general

Using ConditionObject in Android

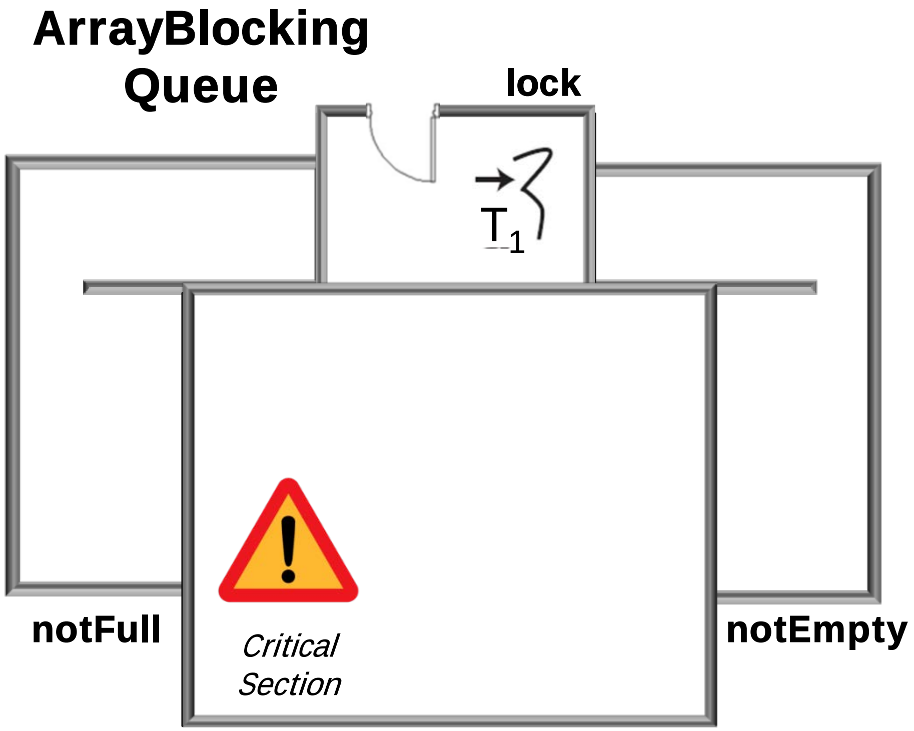
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

```
ArrayBlockingQueue q = new
  ArrayBlockingQueue<String>(10);
...
// Called by thread T1
String s = q.take();
...
```



Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

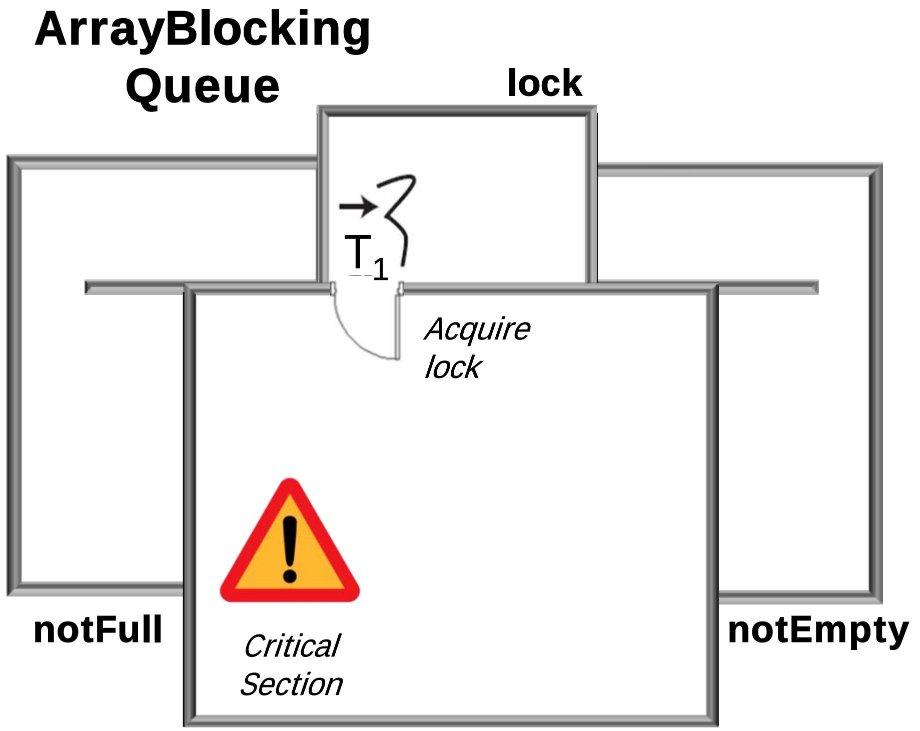


```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
    }
  }
```

Using ConditionObject in Android

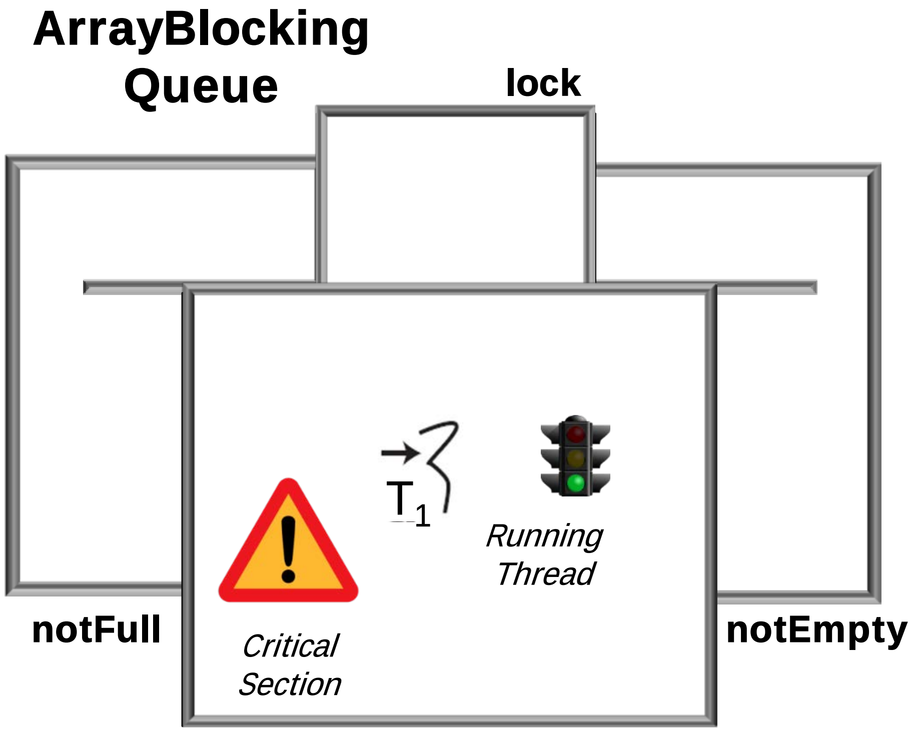
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

```
public class ArrayBlockingQueue<E>
                 extends AbstractQueue<E>
              implements BlockingQueue<E>,
                    java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
    }
  }
```



Using ConditionObject in Android

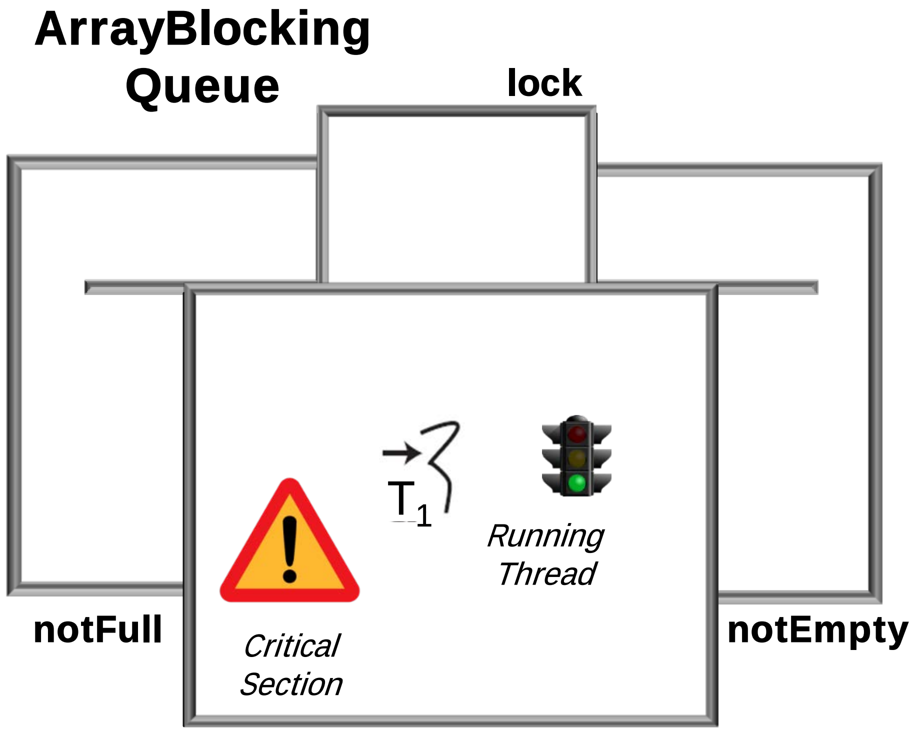
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern



```
public class ArrayBlockingQueue<E>
                extends AbstractQueue<E>
             implements BlockingQueue<E>,
                   java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
    }
  }
```

Using ConditionObject in Android

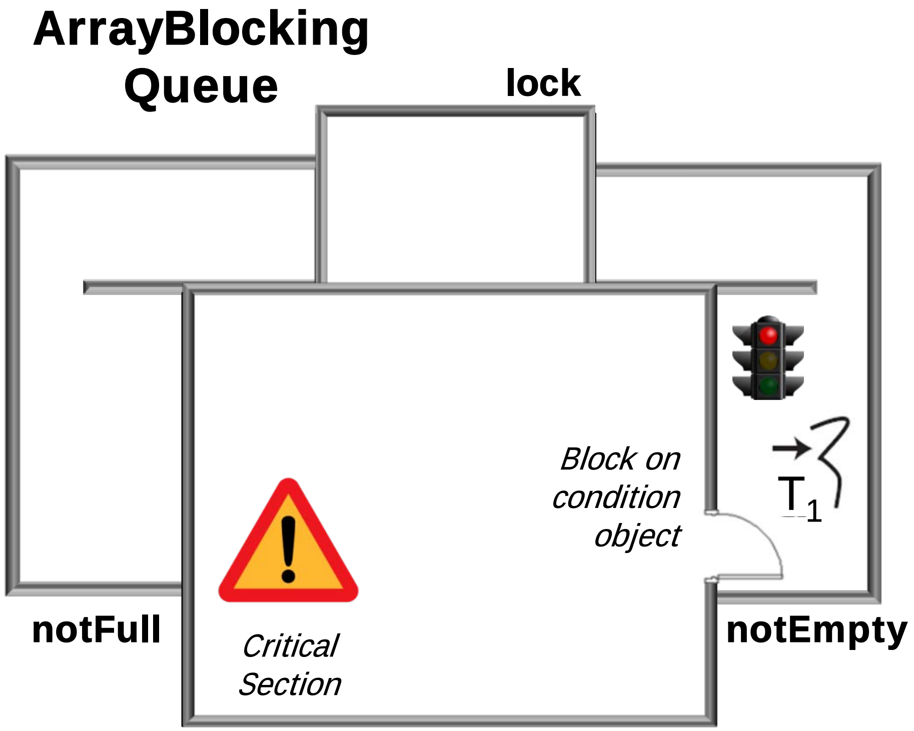
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern



```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
    }
  }
```

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

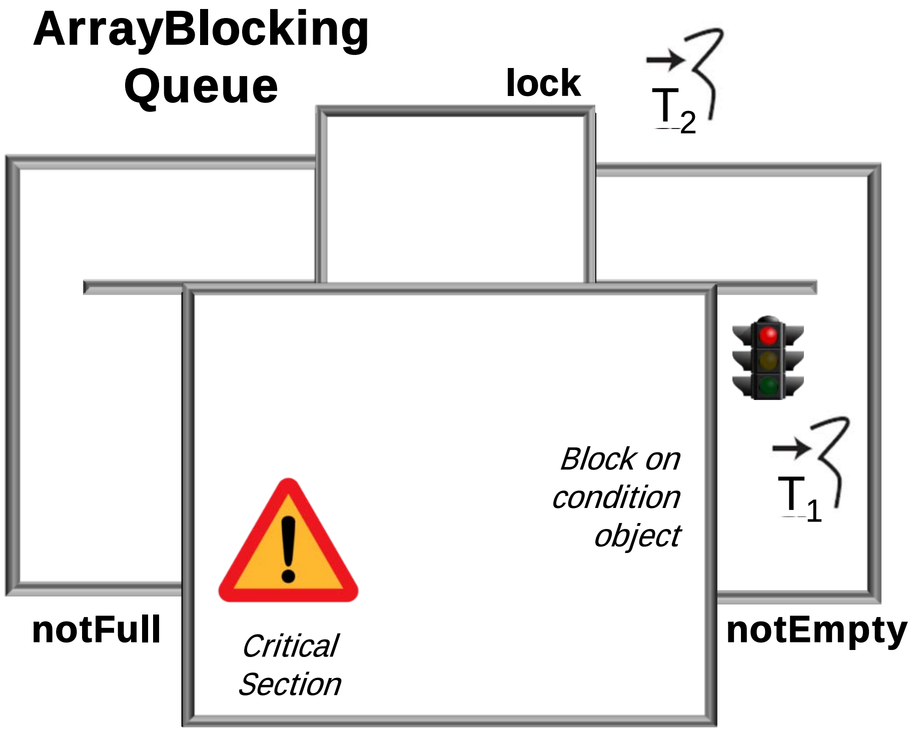


```
public class ArrayBlockingQueue<E>
                extends AbstractQueue<E>
             implements BlockingQueue<E>,
                   java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
    }
  }
```

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

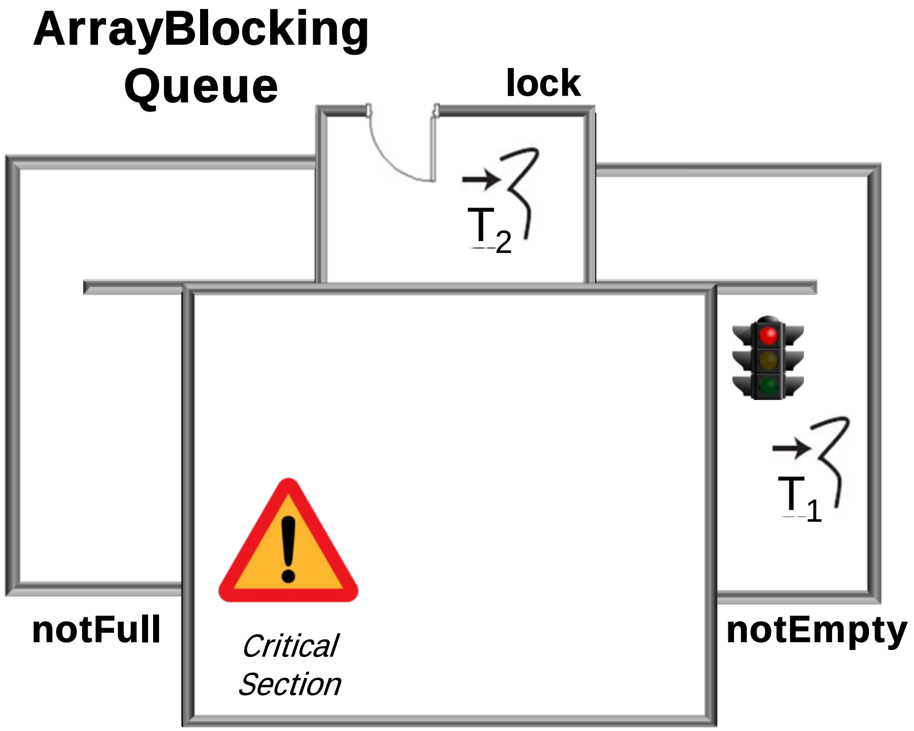
```
...
// Called by thread T2
String s =
  new String("...");
...
q.put(s);
...
```



Using ConditionObject in Android

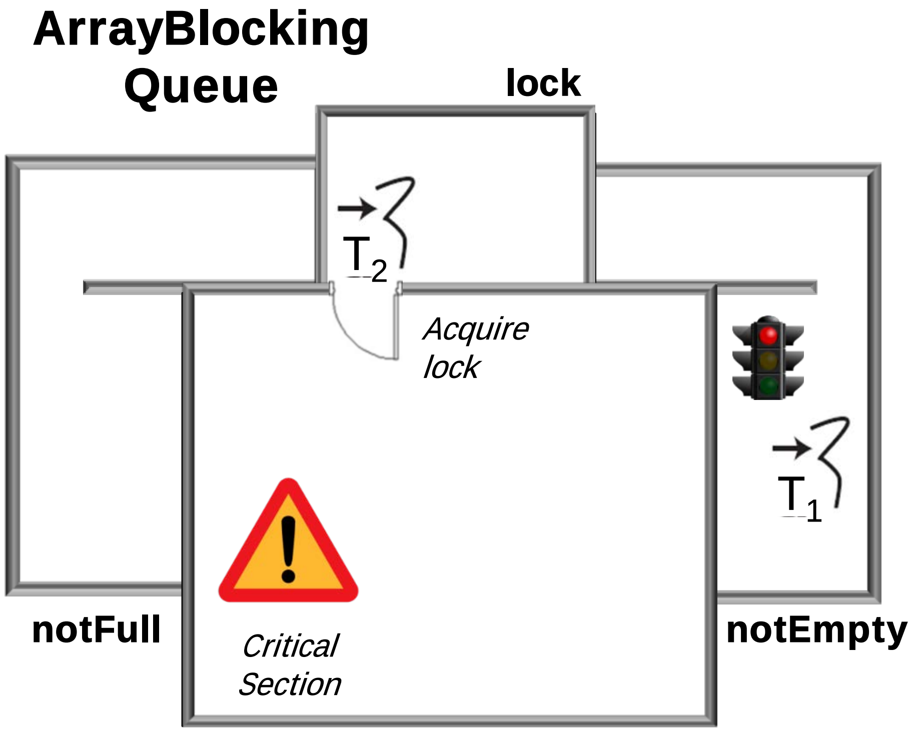
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  public void put(E e) … {
    ...
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == items.length)
        notFull.await();
      insert(e);
    } finally { lock.unlock();
  }
```



Using ConditionObject in Android

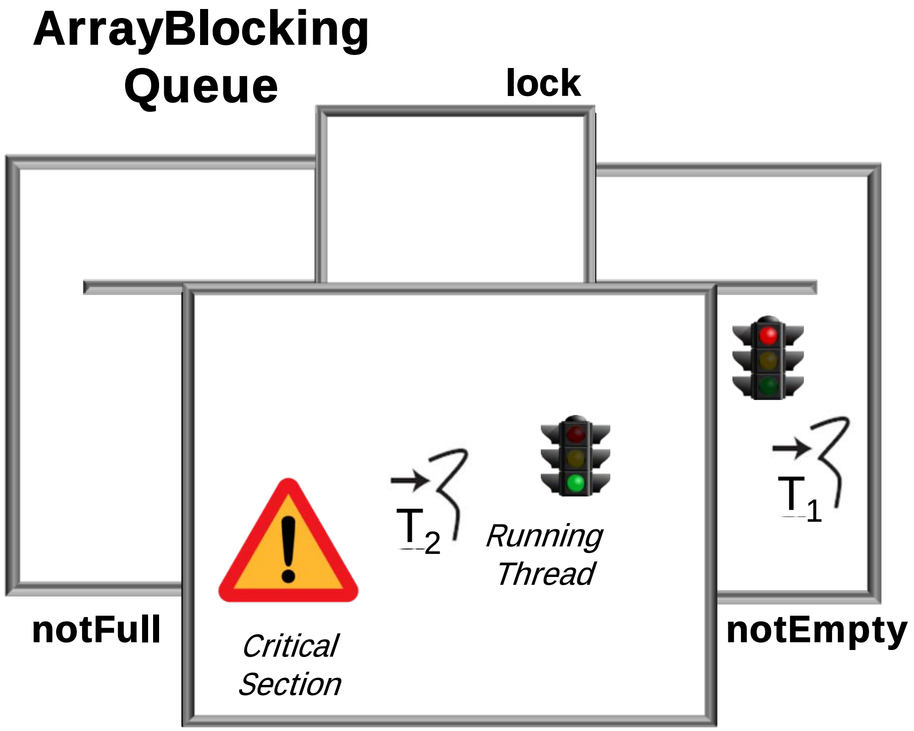
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern



```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  public void put(E e) … {
    ...
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == items.length)
        notFull.await();
      insert(e);
    } finally { lock.unlock(); }
  }
```

Using ConditionObject in Android

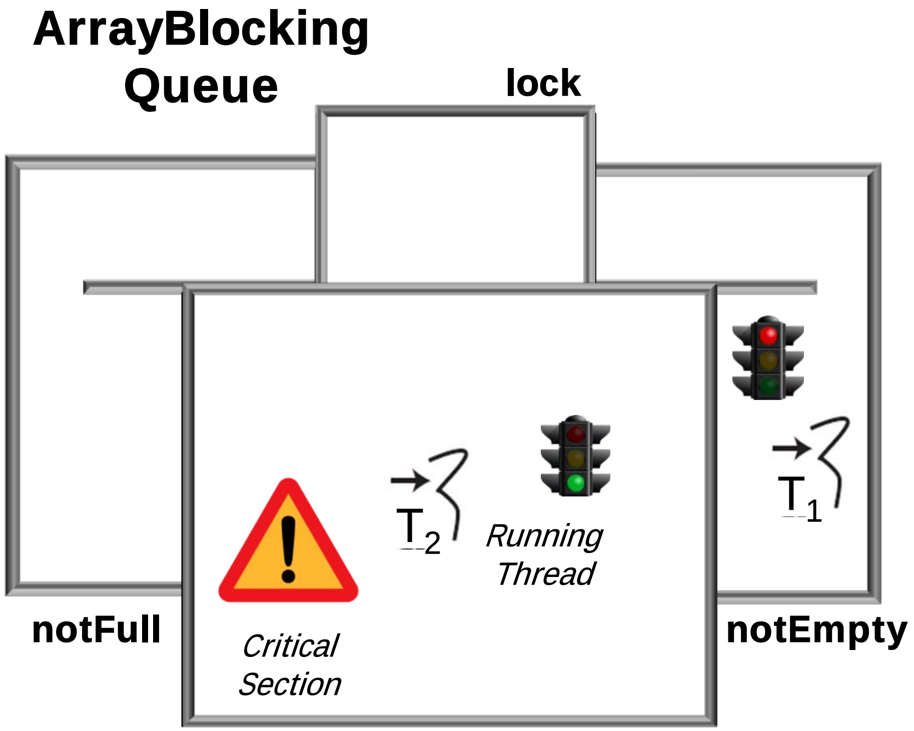
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern



```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  public void put(E e) … {
    ...
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == items.length)
        notFull.await();
      insert(e);
    } finally { lock.unlock(); }
  }
```

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

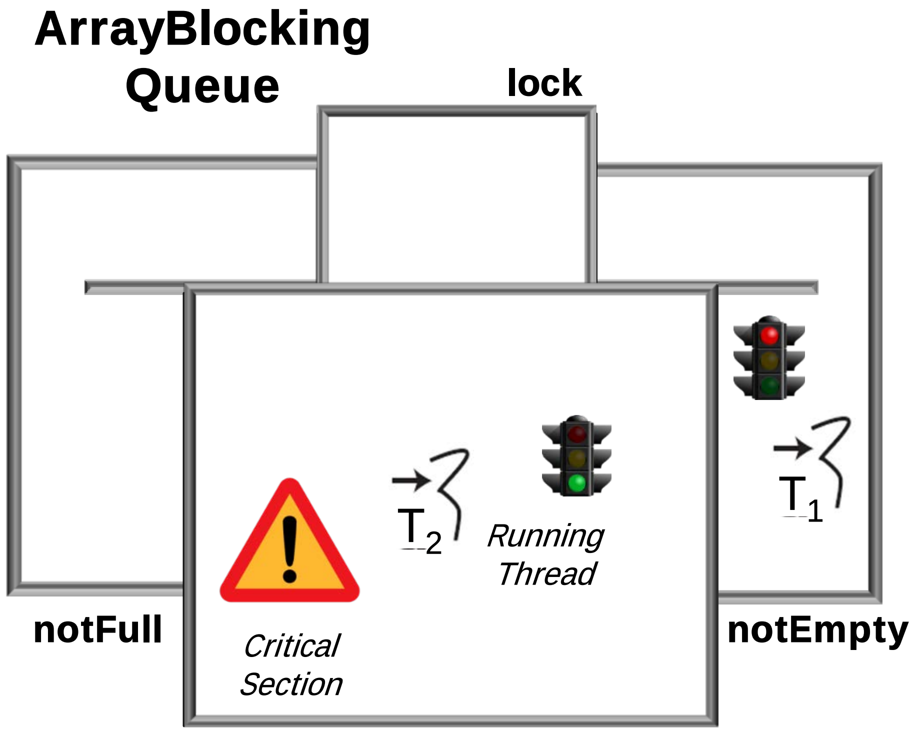


```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  public void put(E e) … {
    ...
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == items.length)
        notFull.await();
      insert(e);
    } finally { lock.unlock(); }
  }
```

Using ConditionObject in Android

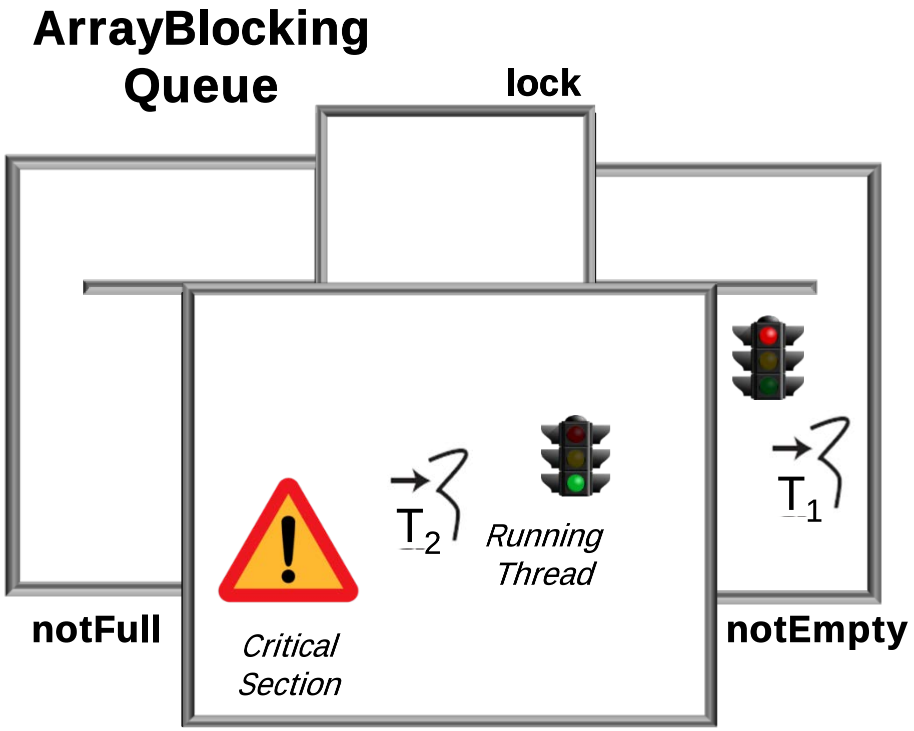
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

```
public class ArrayBlockingQueue<E>
                   extends AbstractQueue<E>
                implements BlockingQueue<E>,
                      java.io.Serializable {
  ...
  private void insert(E x) {
    items[putIndex] = x;
    putIndex = inc(putIndex);
    ++count;
    notEmpty.signal();
  }
```



Using ConditionObject in Android

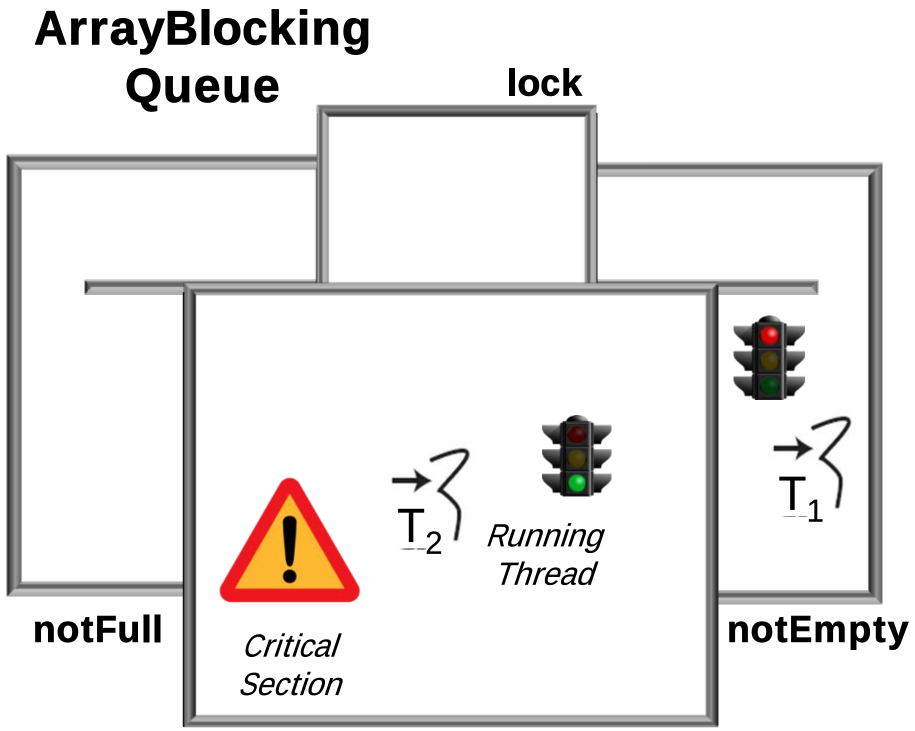
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern



```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  private void insert(E x) {
    items[putIndex] = x;
    putIndex = inc(putIndex);
    ++count;
    notEmpty.signal();
  }
```

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

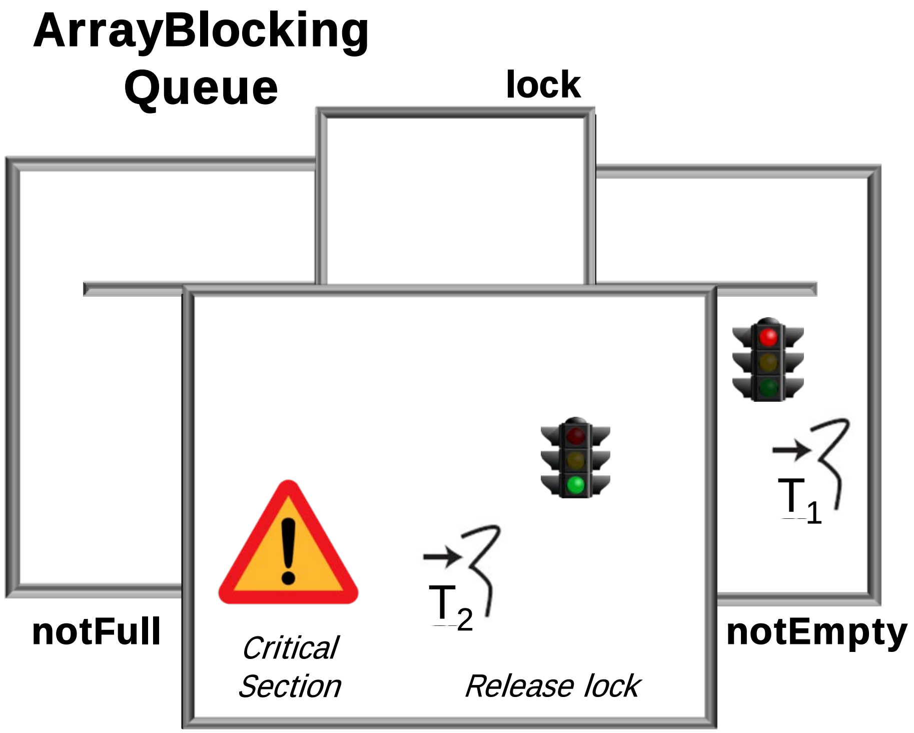


```
public class ArrayBlockingQueue<E>
                    extends AbstractQueue<E>
                 implements BlockingQueue<E>,
                       java.io.Serializable {
  ...
  private void insert(E x) {
    items[putIndex] = x;
    putIndex = inc(putIndex);
    ++count;
    notEmpty.signal();
  }
```

Using ConditionObject in Android

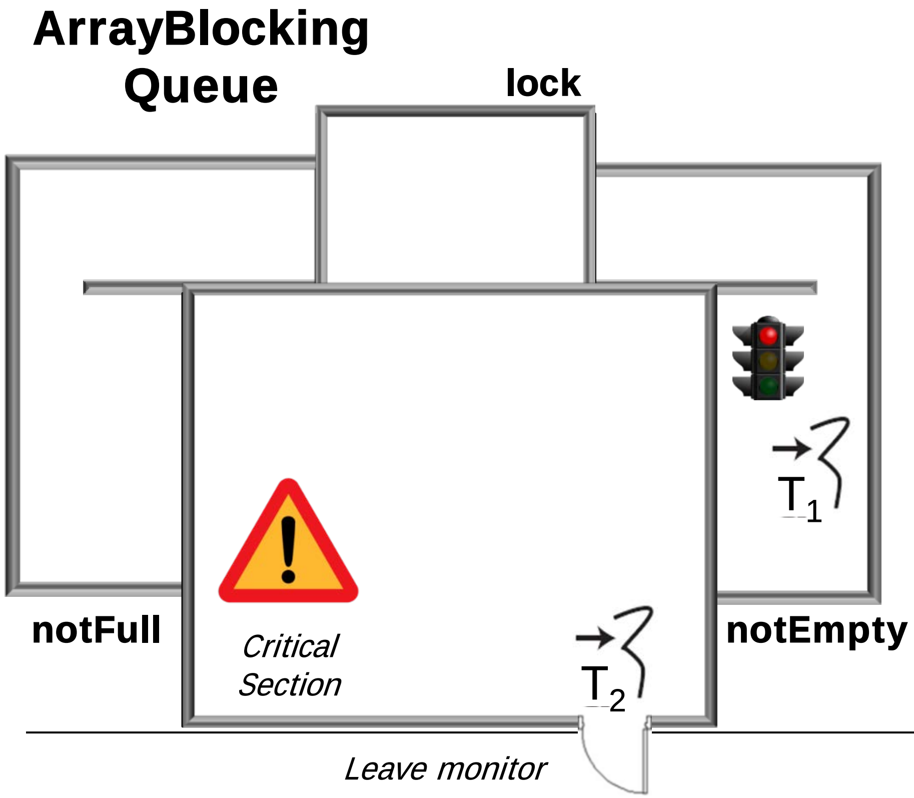
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  public void put(E e) … {
    ...
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == items.length)
        notFull.await();
      insert(e);
    } finally { lock.unlock(); }
  }
```



Using ConditionObject in Android

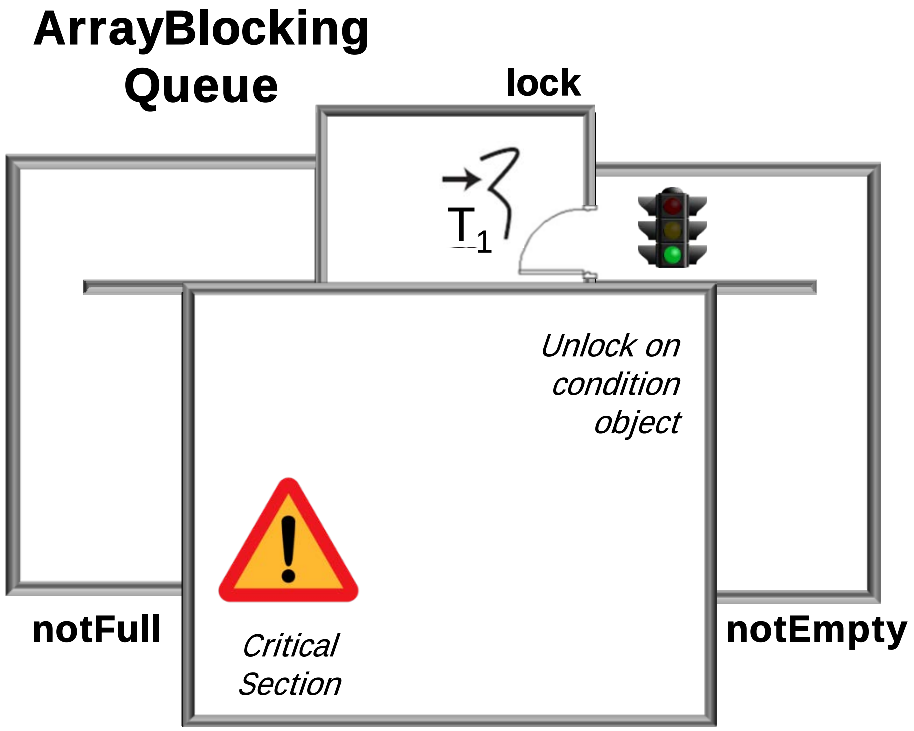
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern



```
public class ArrayBlockingQueue<E>
                 extends AbstractQueue<E>
              implements BlockingQueue<E>,
                    java.io.Serializable {
  ...
  public void put(E e) … {
    ...
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == items.length)
        notFull.await();
      insert(e);
    } finally { lock.unlock(); }
  }
```

Using ConditionObject in Android

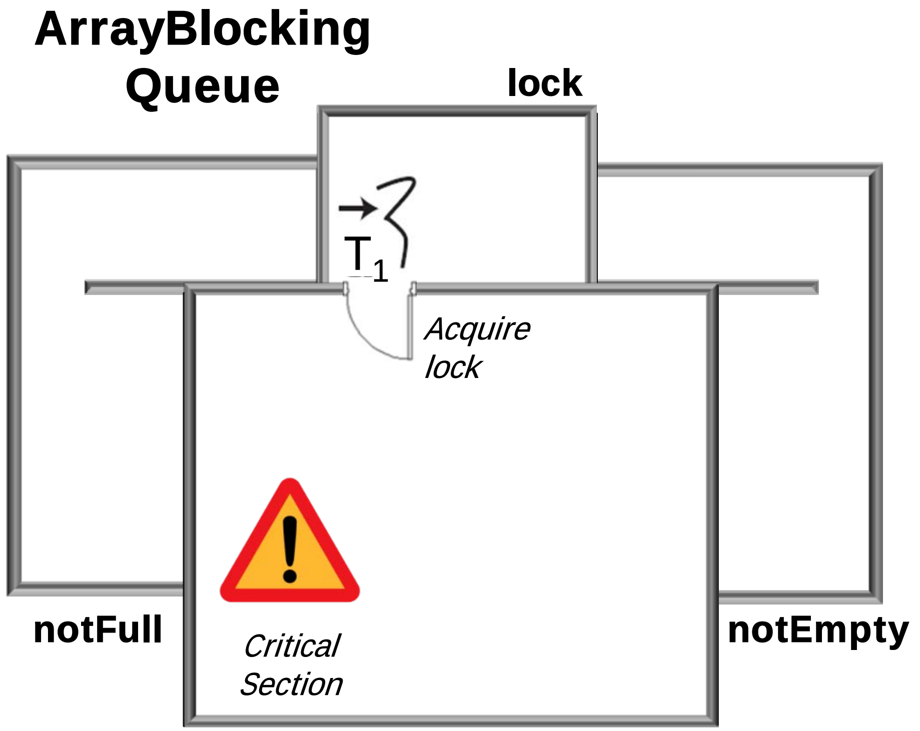
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern



```
public class ArrayBlockingQueue<E>
                extends AbstractQueue<E>
             implements BlockingQueue<E>,
                   java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
    }
  }
```

Using ConditionObject in Android

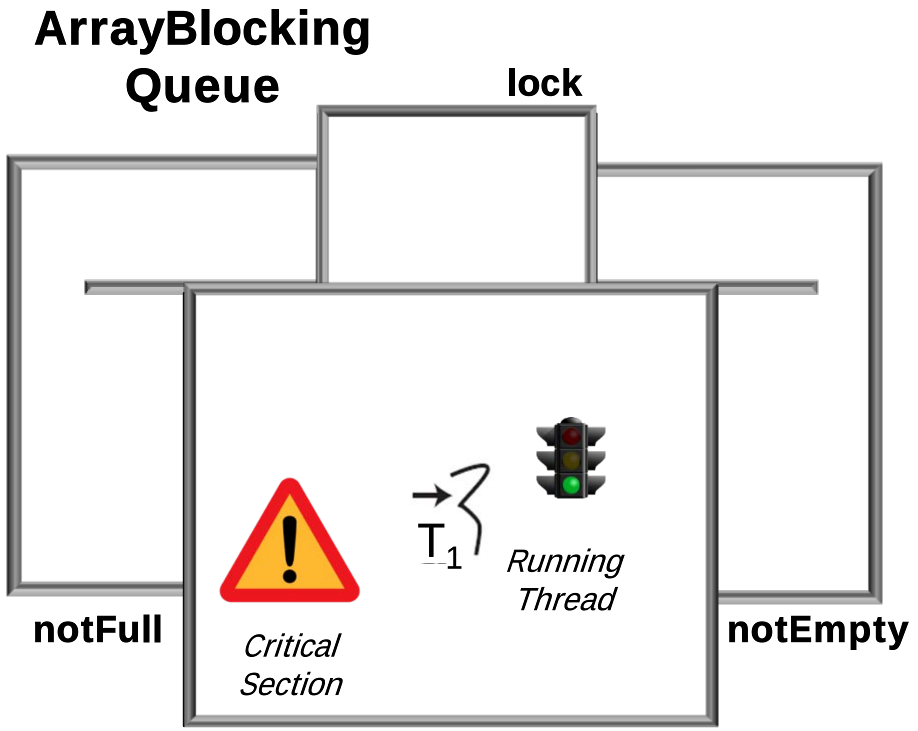
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern



```
public class ArrayBlockingQueue<E>
          extends AbstractQueue<E>
       implements BlockingQueue<E>,
             java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
    }
  }
```

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

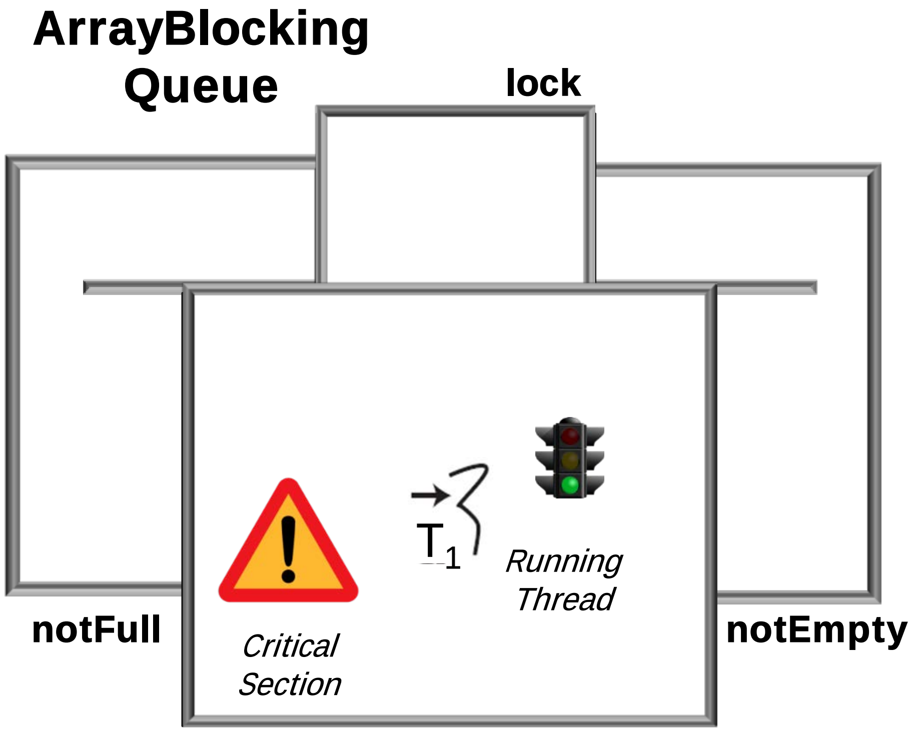


```
public class ArrayBlockingQueue<E>
                  extends AbstractQueue<E>
               implements BlockingQueue<E>,
                     java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
    }
  }
```

Using ConditionObject in Android

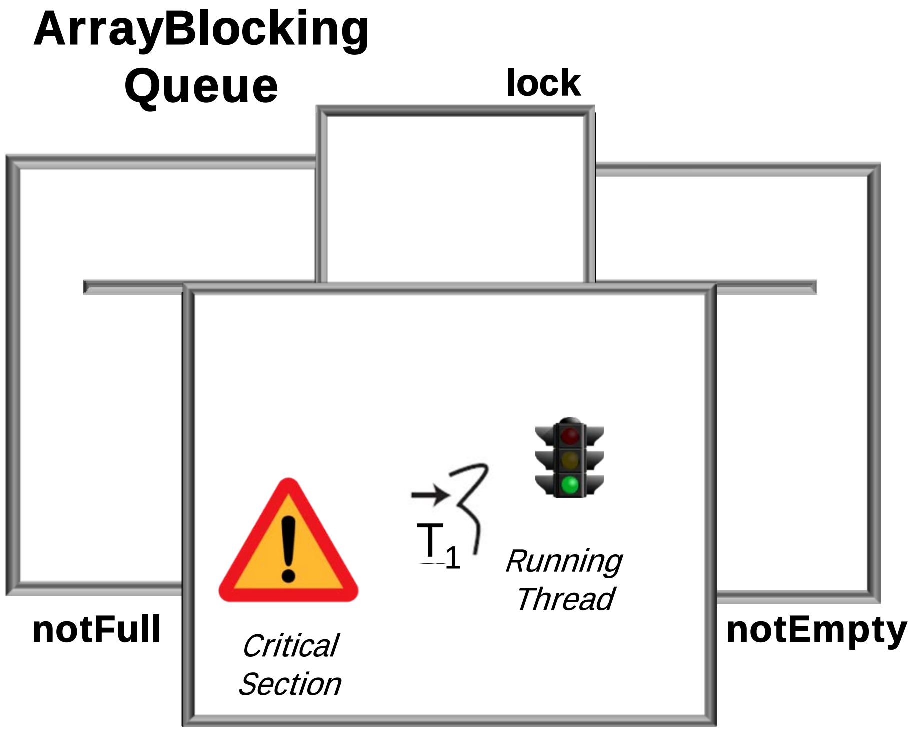
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
    }
  }
```



Using ConditionObject in Android

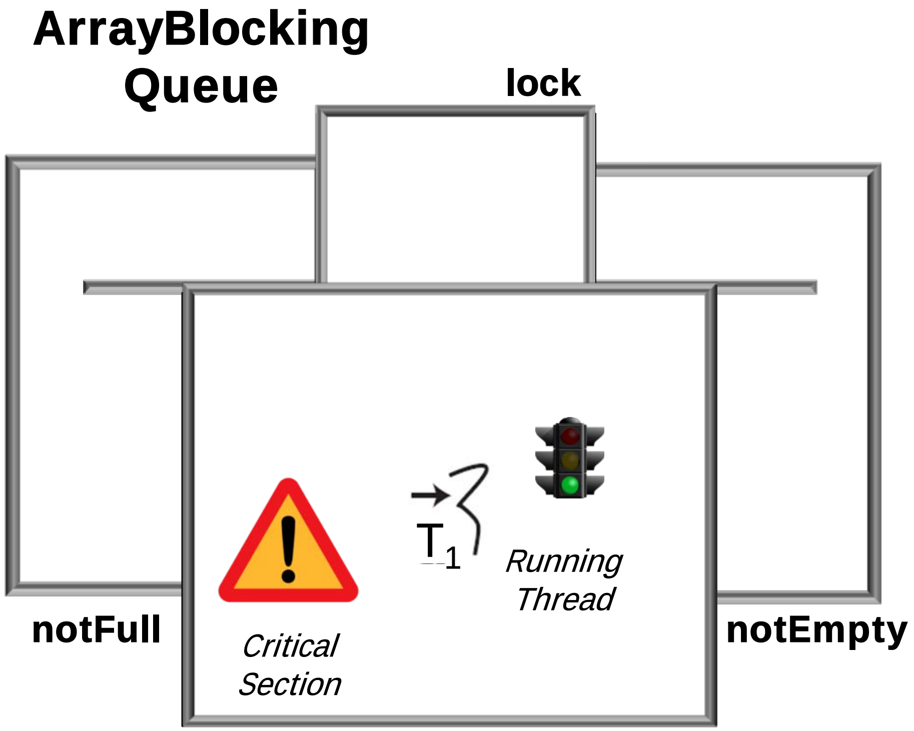
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern



```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  private E extract() {
    final Object[] items =
      this.items;
    E x =
      this.<E>cast
        (items[takeIndex]);
    items[takeIndex] = null;
    takeIndex = inc(takeIndex);
    --count;
    notFull.signal();
    return x;
  }
```

Using ConditionObject in Android

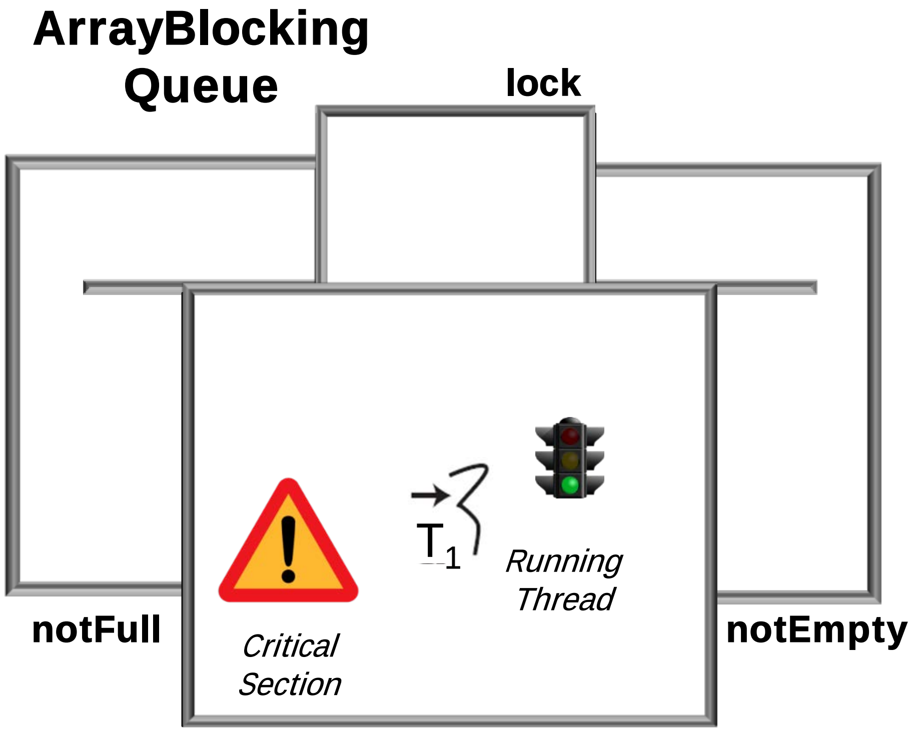
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern



```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  private E extract() {
    final Object[] items =
      this.items;
    E x =
      this.<E>cast
        (items[takeIndex]);
    items[takeIndex] = null;
    takeIndex = inc(takeIndex);
    --count;
    notFull.signal();
    return x;
  }
```

Using ConditionObject in Android

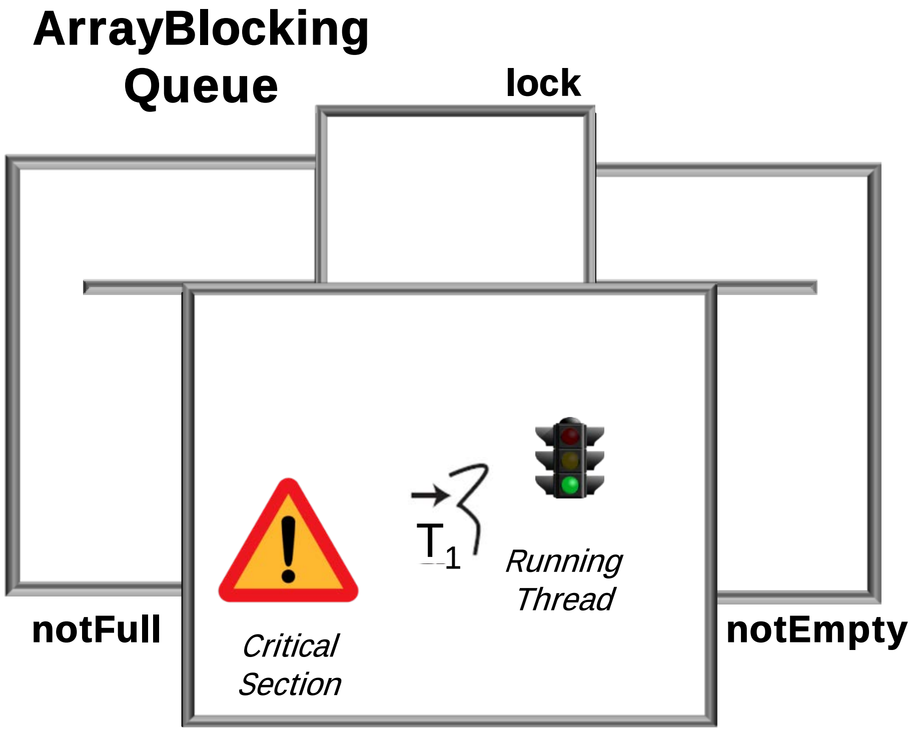
- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern



```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  private E extract() {
    final Object[] items =
      this.items;
    E x =
      this.<E>cast
        (items[takeIndex]);
    items[takeIndex] = null;
    takeIndex = inc(takeIndex);
    --count;
    notFull.signal();
    return x;
  }
```

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

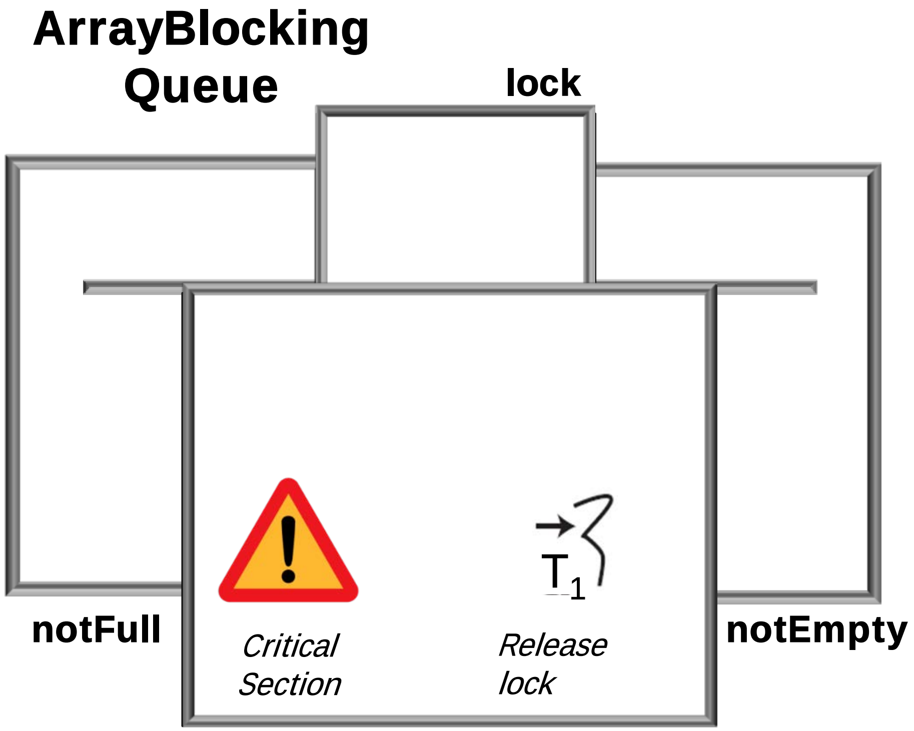


```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  private E extract() {
    final Object[] items =
      this.items;
    E x =
      this.<E>cast
        (items[takeIndex]);
    items[takeIndex] = null;
    takeIndex = inc(takeIndex);
    --count;
    notFull.signal();
    return x;
  }
```

Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

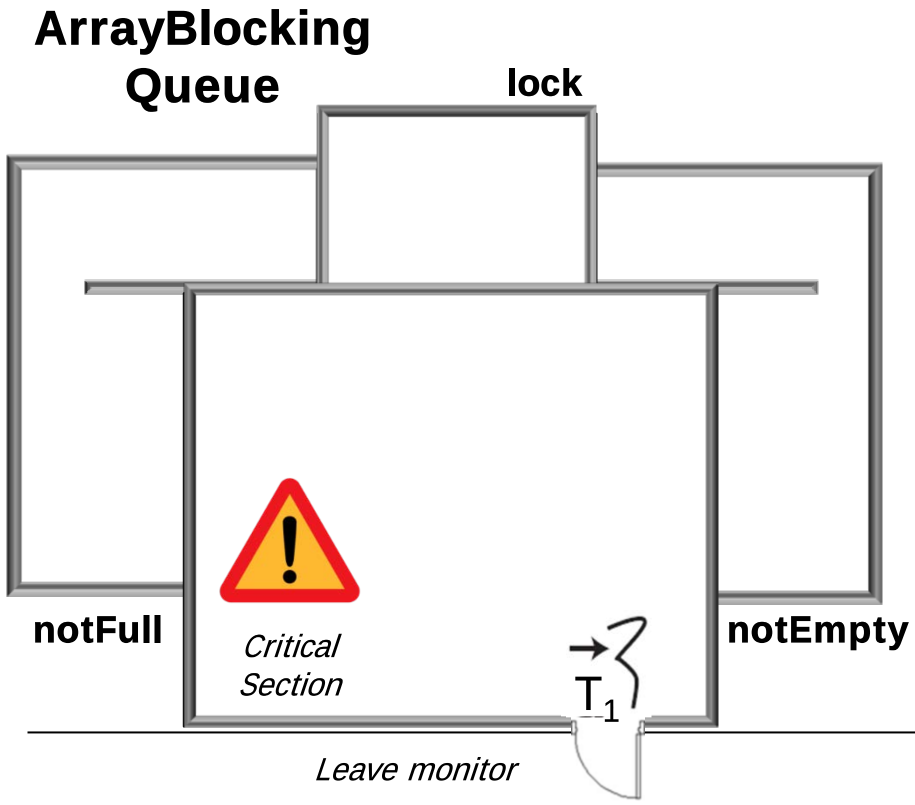
```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
    }
  }
```



Using ConditionObject in Android

- ArrayBlockingQueue is a blocking bounded FIFO queue
- ReentrantLock & Condition Objects implement the *Monitor Object* pattern

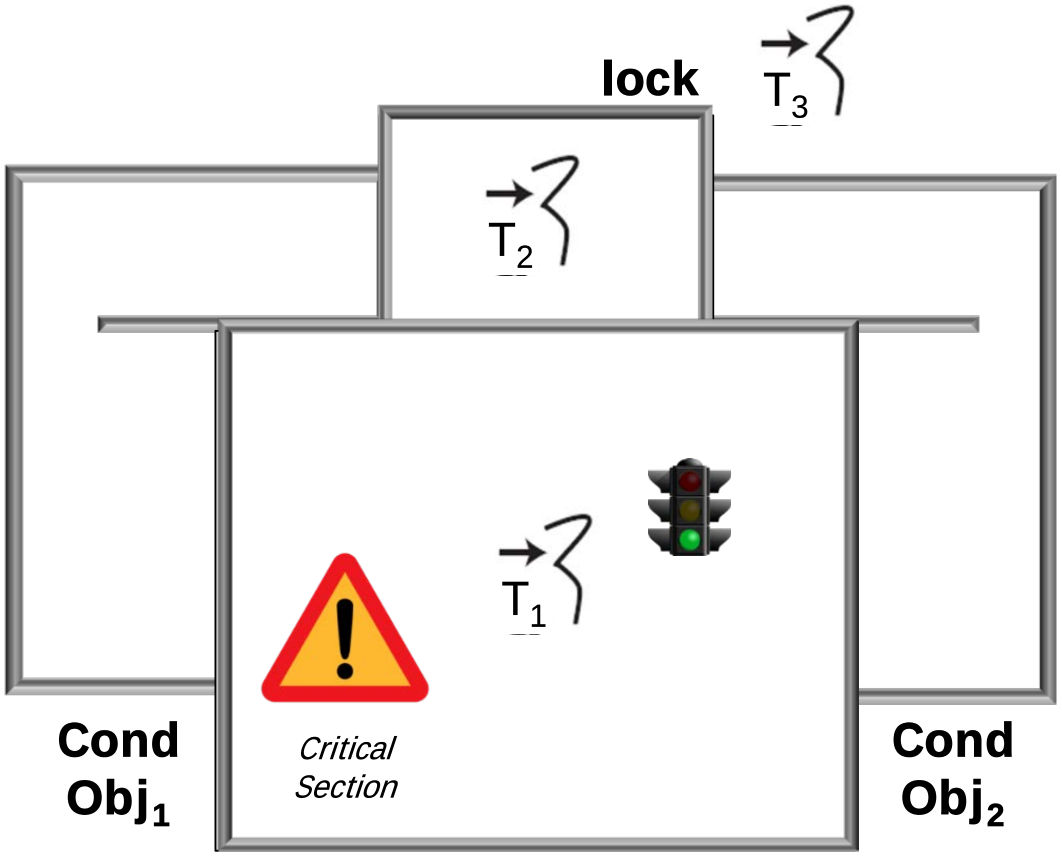
```
public class ArrayBlockingQueue<E>
           extends AbstractQueue<E>
        implements BlockingQueue<E>,
              java.io.Serializable {
  ...
  public E take() ... {
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
    }
  }
```



Summary

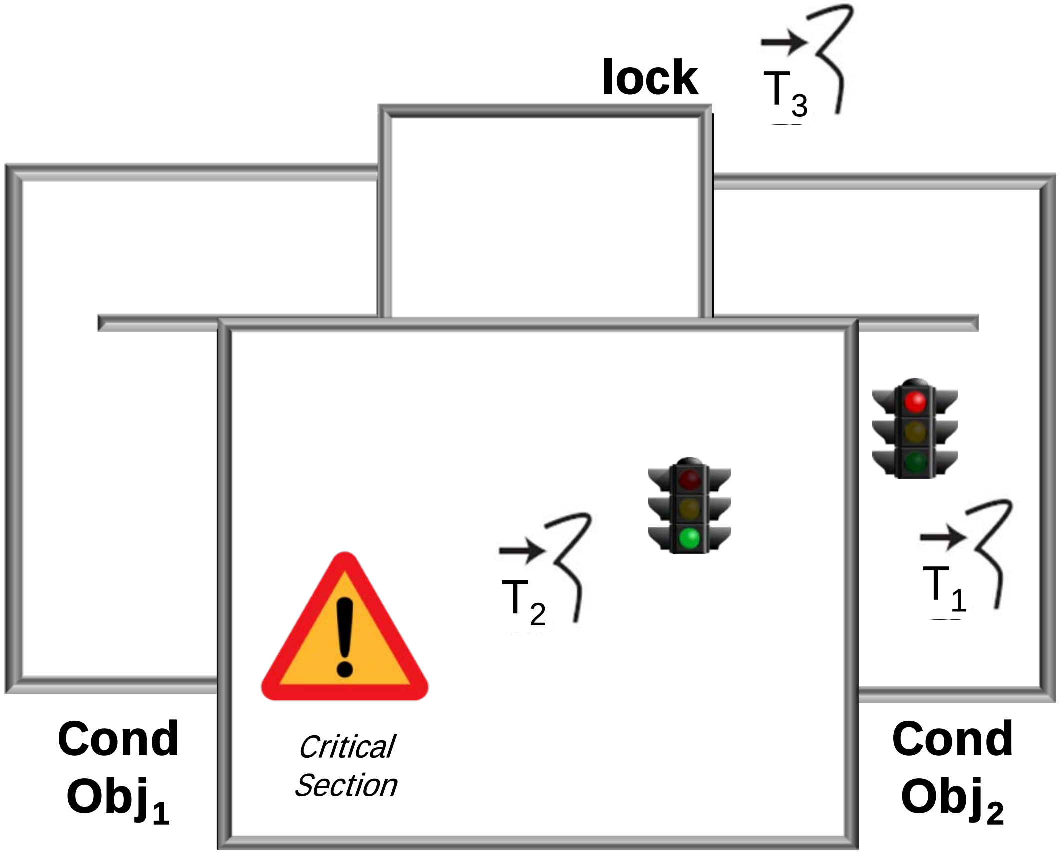
Summary

- A ConditionObject provides a flexible synchronization & scheduling mechanism



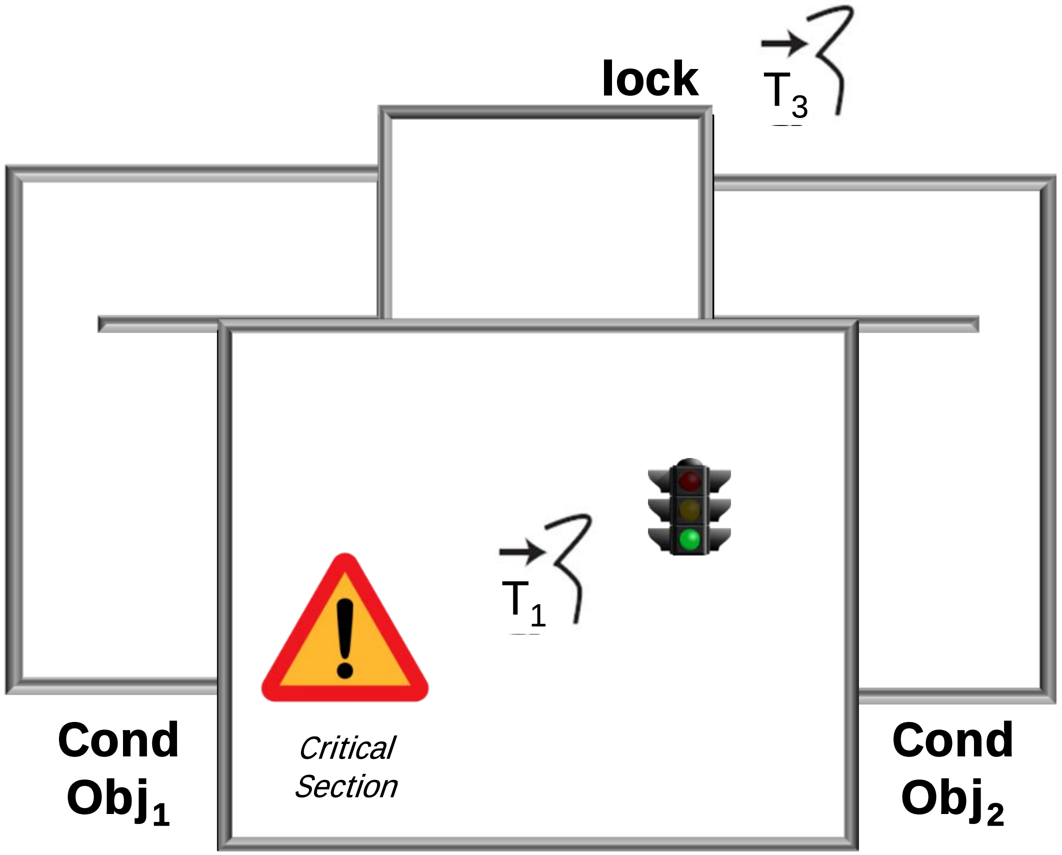
Summary

- A ConditionObject provides a flexible synchronization & scheduling mechanism
  - Allows threads to suspend & resume their execution



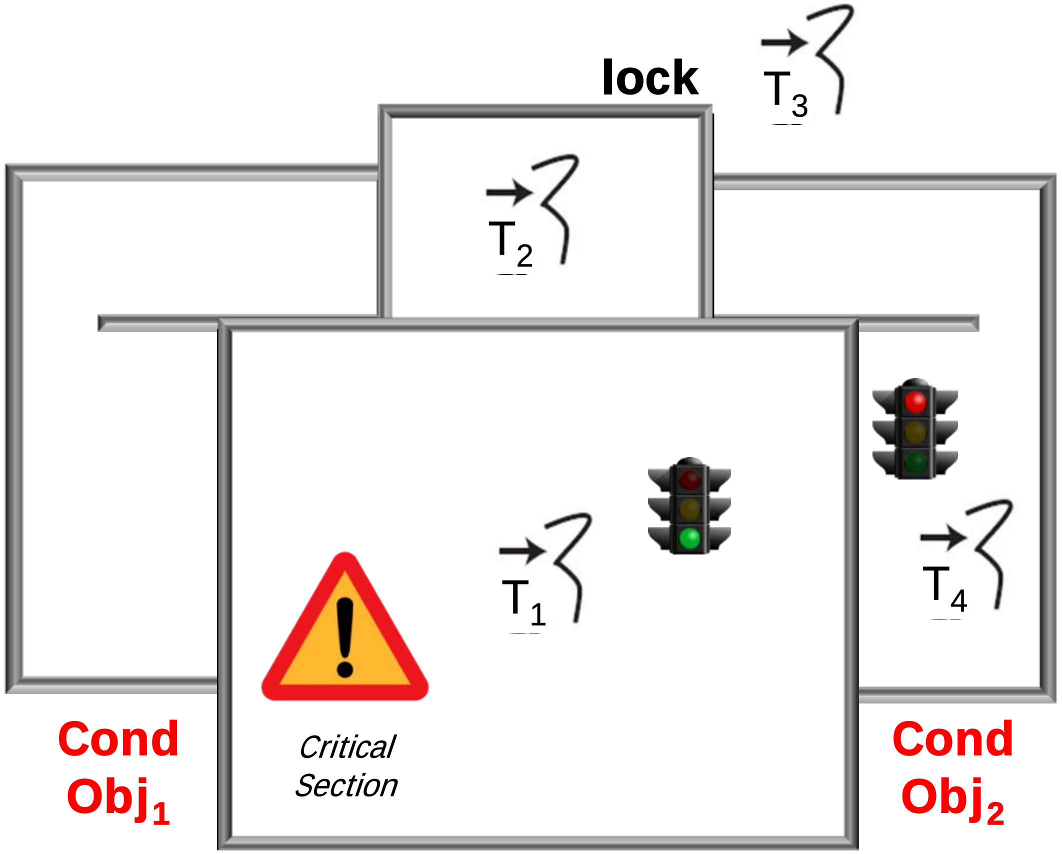
Summary

- A ConditionObject provides a flexible synchronization & scheduling mechanism
  - Allows threads to suspend & resume their execution



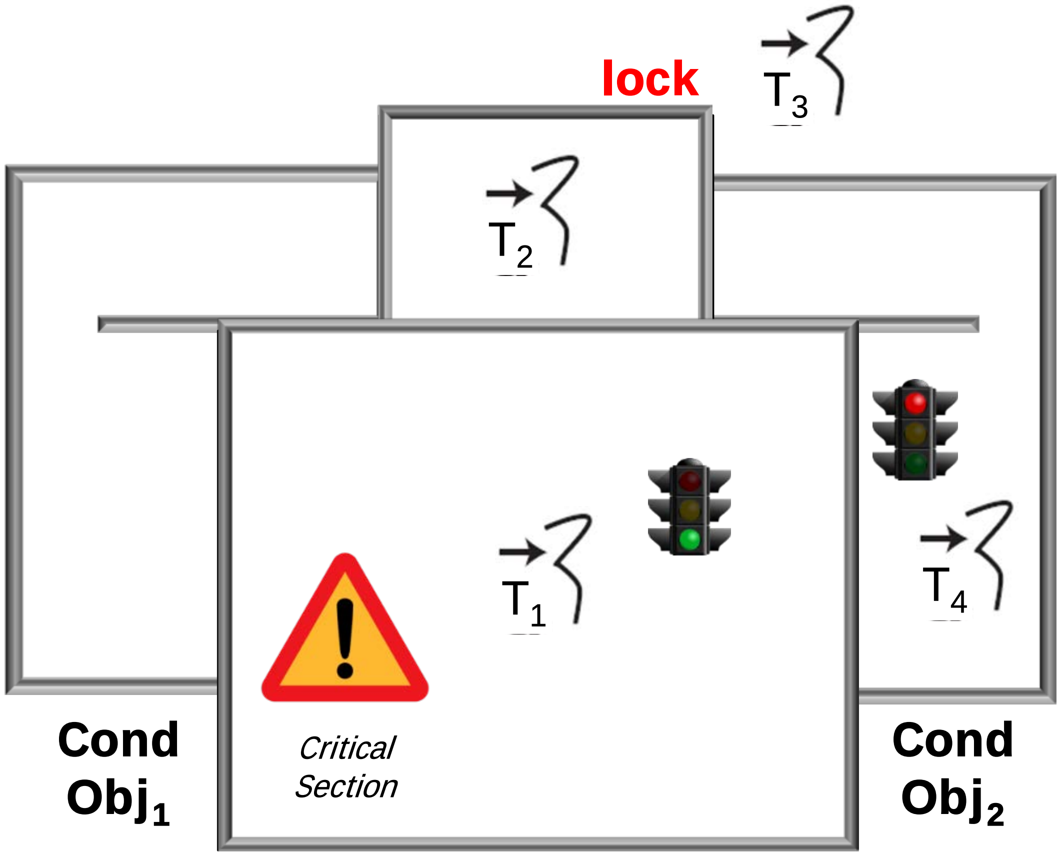
Summary

- A ConditionObject provides a flexible synchronization & scheduling mechanism
  - Allows threads to suspend & resume their execution
  - An object can have multiple ConditionObjects



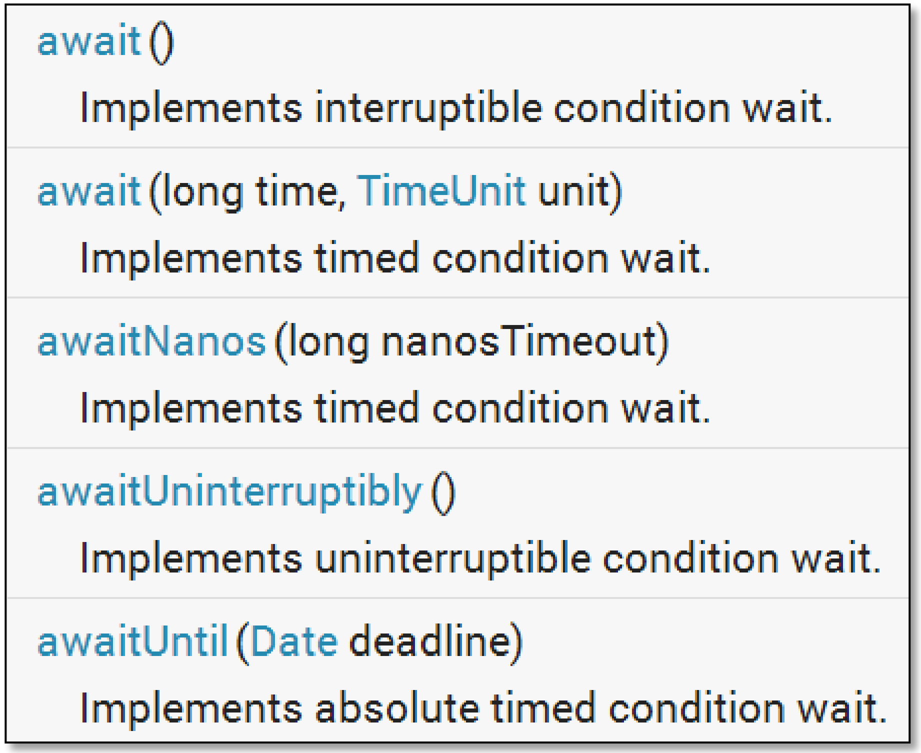
Summary

- A ConditionObject provides a flexible synchronization & scheduling mechanism
  - Allows threads to suspend & resume their execution
  - An object can have multiple ConditionObjects
  - It is always used in conjunction with a lock



Summary

- A ConditionObject provides a flexible synchronization & scheduling mechanism
  - Allows threads to suspend & resume their execution
  - An object can have multiple ConditionObjects
  - It is always used in conjunction with a lock
  - It supports several types of wait operations
    - e.g., interruptible, non-interruptible, & timed operations



await()	Implements interruptible condition wait.
await(long time, TimeUnit unit)	Implements timed condition wait.
awaitNanos(long nanosTimeout)	Implements timed condition wait.
awaitUninterruptibly()	Implements uninterruptible condition wait.
awaitUntil(Date deadline)	Implements absolute timed condition wait.

Summary

- A ConditionObject provides a flexible synchronization & scheduling mechanism
  - Allows threads to suspend & resume their execution
  - An object can have multiple ConditionObjects
  - It is always used in conjunction with a lock
  - It supports several types of wait operations
  - It should always be waited upon in a loop

```
public class
          ArrayBlockingQueue<E>
                            ... {
  ...
  public E take() ... {
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
    }
  }
```

Summary

- A ConditionObject provides a flexible synchronization & scheduling mechanism
  - Allows threads to suspend & resume their execution
  - An object can have multiple ConditionObjects
  - It is always used in conjunction with a lock
  - It supports several types of wait operations
  - It should always be waited upon in a loop
    - Test state predicate being waited for

```
public class
         ArrayBlockingQueue<E>
                          ... {
  ...
  public E take() ... {
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
    }
  }
```

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

Summary

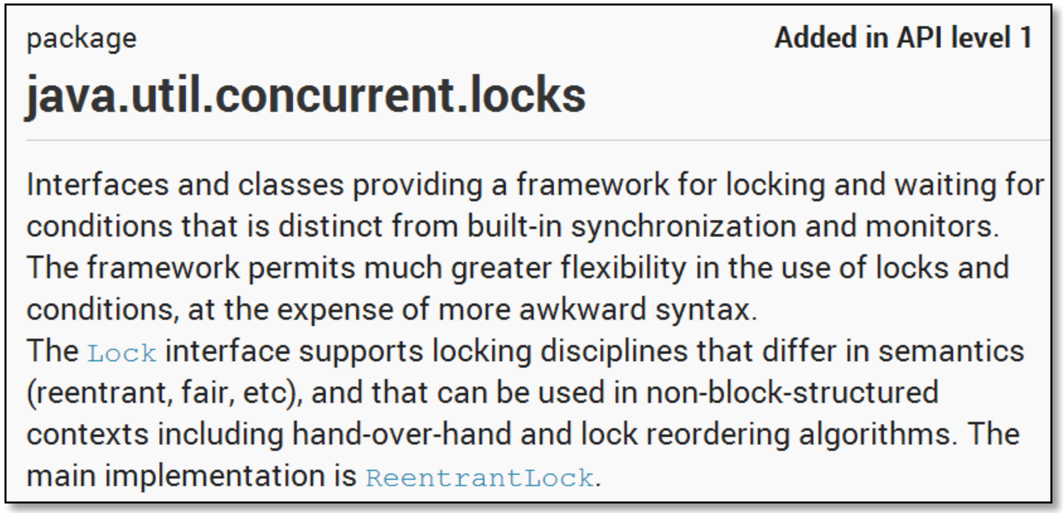
- A ConditionObject provides a flexible synchronization & scheduling mechanism
  - Allows threads to suspend & resume their execution
  - An object can have multiple ConditionObjects
  - It is always used in conjunction with a lock
  - It supports several types of wait operations
  - It should always be waited upon in a loop
    - Test state predicate being waited for
    - Guard against spurious wakeups

```
public class
          ArrayBlockingQueue<E>
                          ... {
  ...
  public E take() ... {
    final ReentrantLock lock =
      this.lock;
    lock.lockInterruptibly();
    try {
      while (count == 0)
        notEmpty.await();
      return extract();
    } finally {
      lock.unlock();
    }
  }
```

en.wikipedia.org/wiki/Spurious_wakeup has more info

Summary

- A ConditionObject provides a flexible synchronization & scheduling mechanism
- ConditionObject is used sparingly in Android
  - Mostly in java.util.concurrent & java.util.concurrent.locks

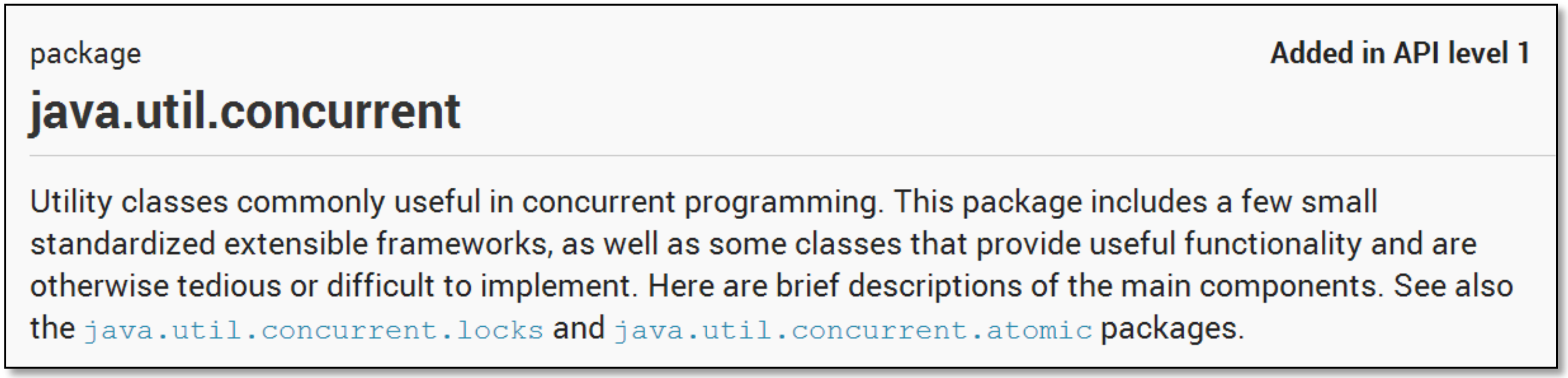

package

Added in API level 1

java.util.concurrent.locks

Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors. The framework permits much greater flexibility in the use of locks and conditions, at the expense of more awkward syntax.

The `Lock` interface supports locking disciplines that differ in semantics (reentrant, fair, etc), and that can be used in non-block-structured contexts including hand-over-hand and lock reordering algorithms. The main implementation is `ReentrantLock`.


package

Added in API level 1

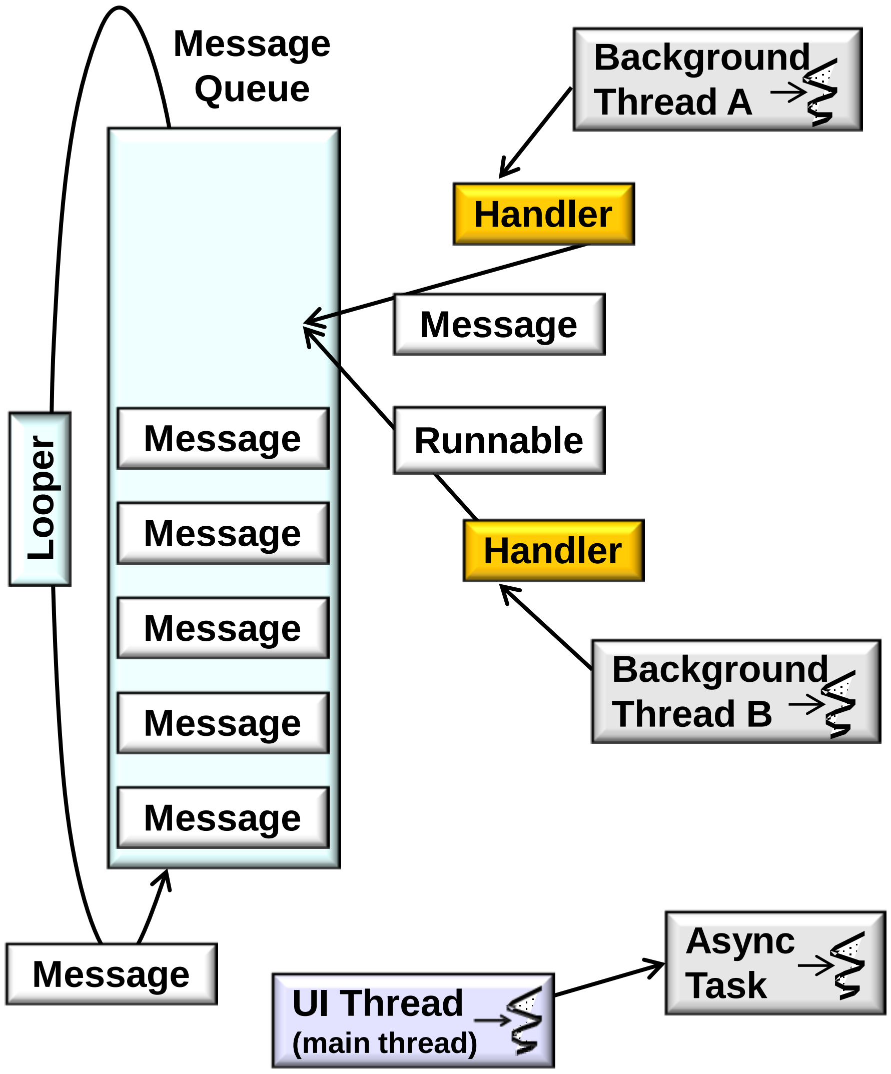
java.util.concurrent

Utility classes commonly useful in concurrent programming. This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement. Here are brief descriptions of the main components. See also the `java.util.concurrent.locks` and `java.util.concurrent.atomic` packages.

libcore/luni/src/main/java/java/util/concurrent contains all the source code

Summary

- A ConditionObject provides a flexible synchronization & scheduling mechanism
- ConditionObject is used sparingly in Android
  - Mostly in java.util.concurrent & java.util.concurrent.locks
  - Thus used in Android's concurrency frameworks



See upcoming module on "Android Concurrency Frameworks"