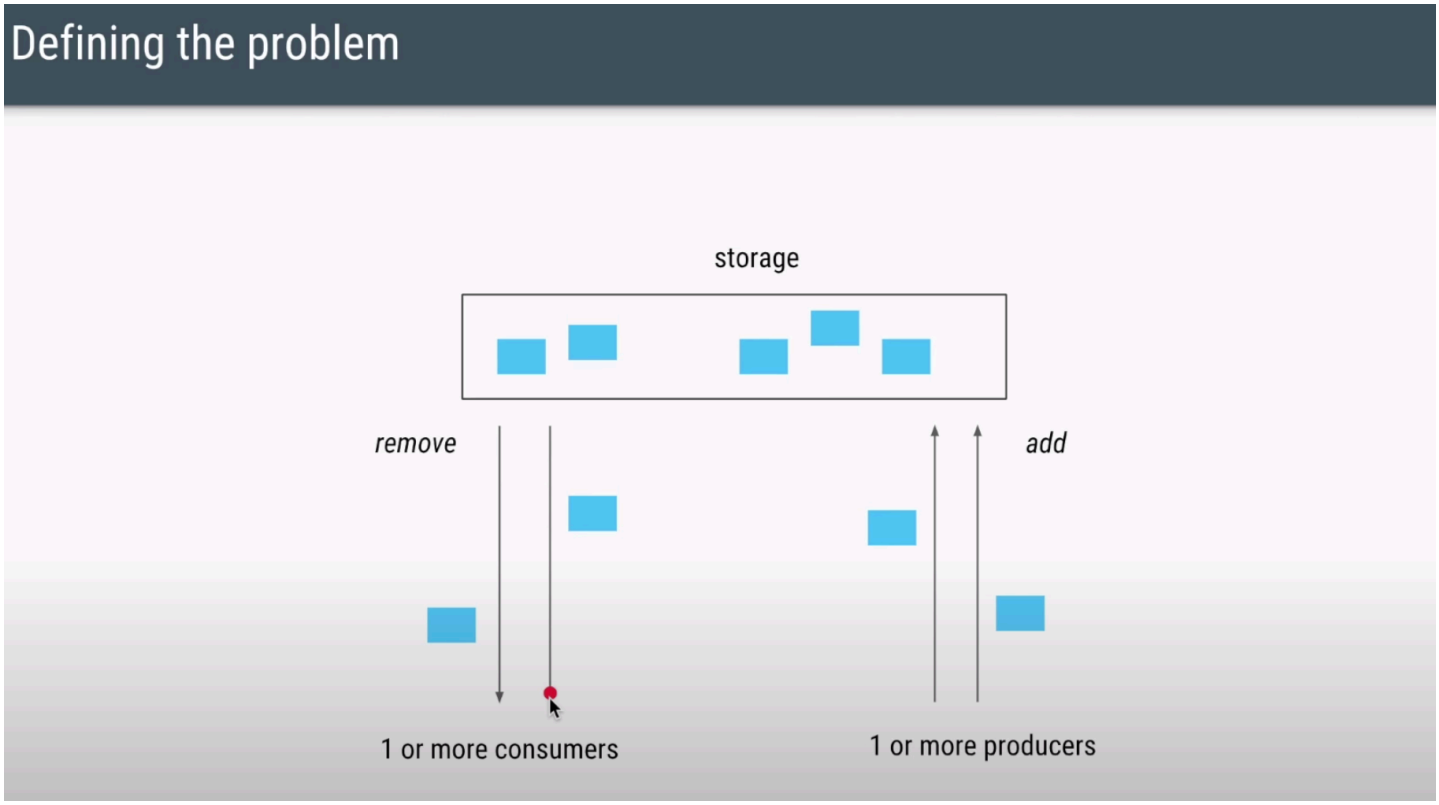
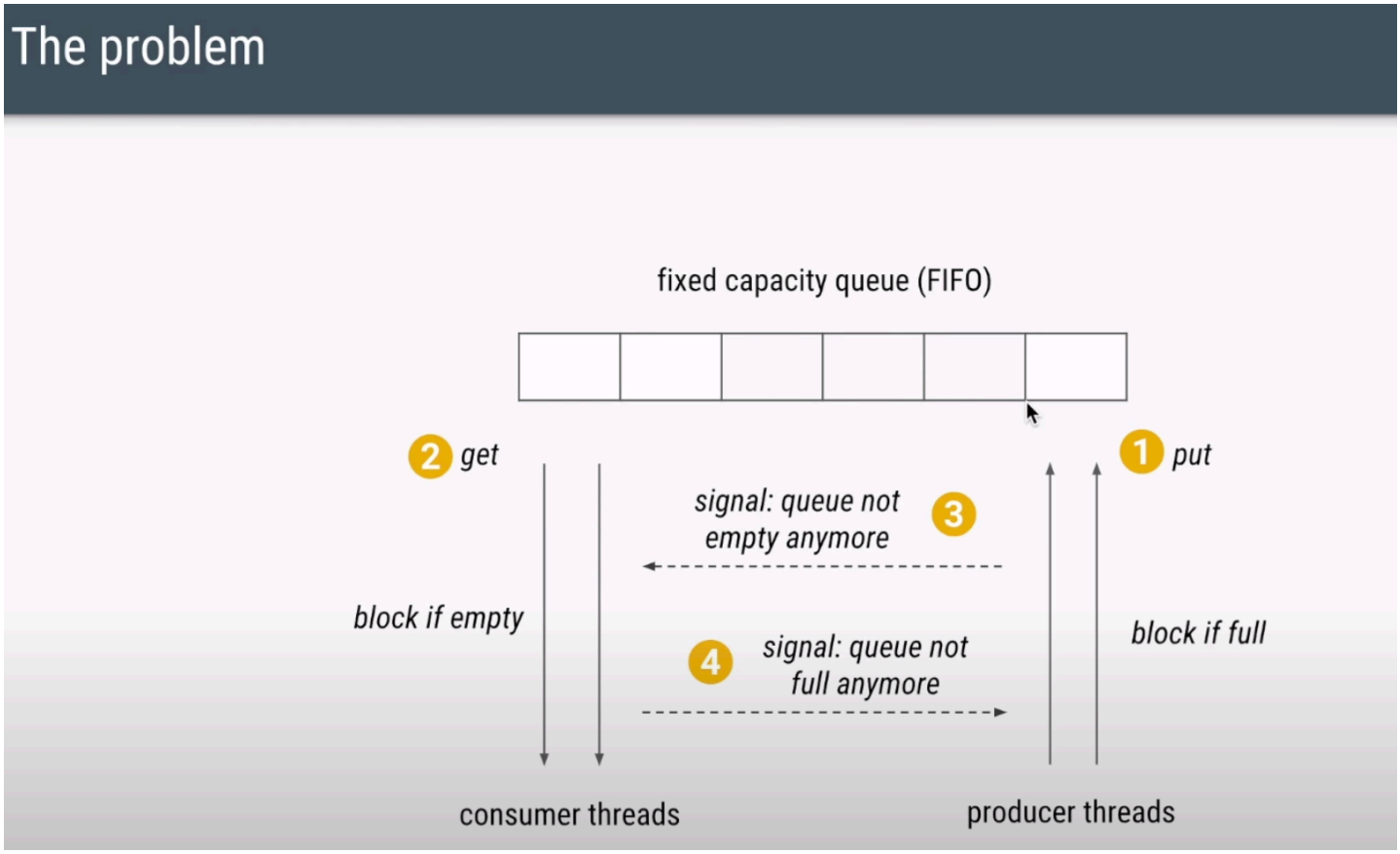
Multithreading Interview Question Part1: Implement Producer-Consumer pattern

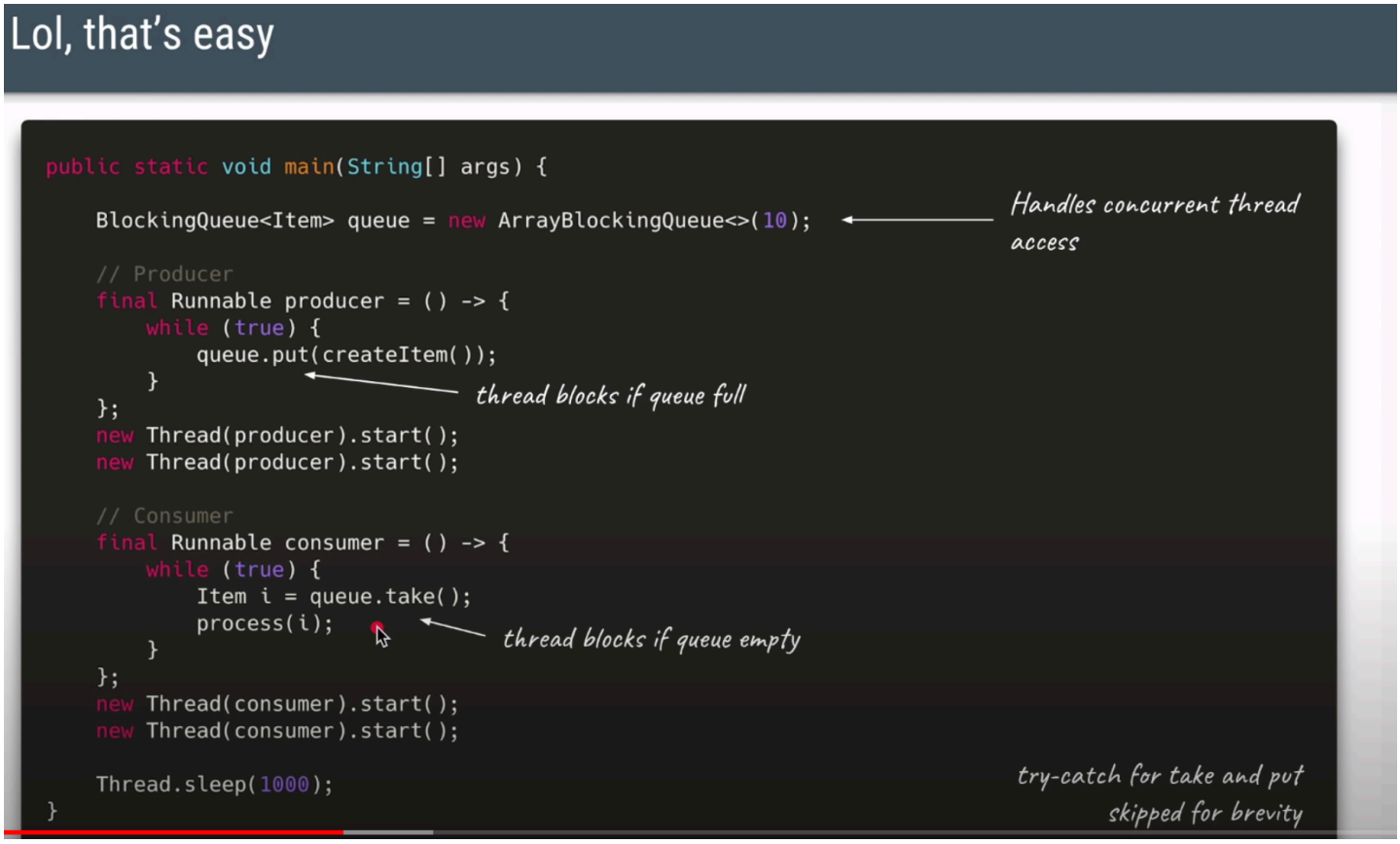


→ **Consumer blocked if no item present**

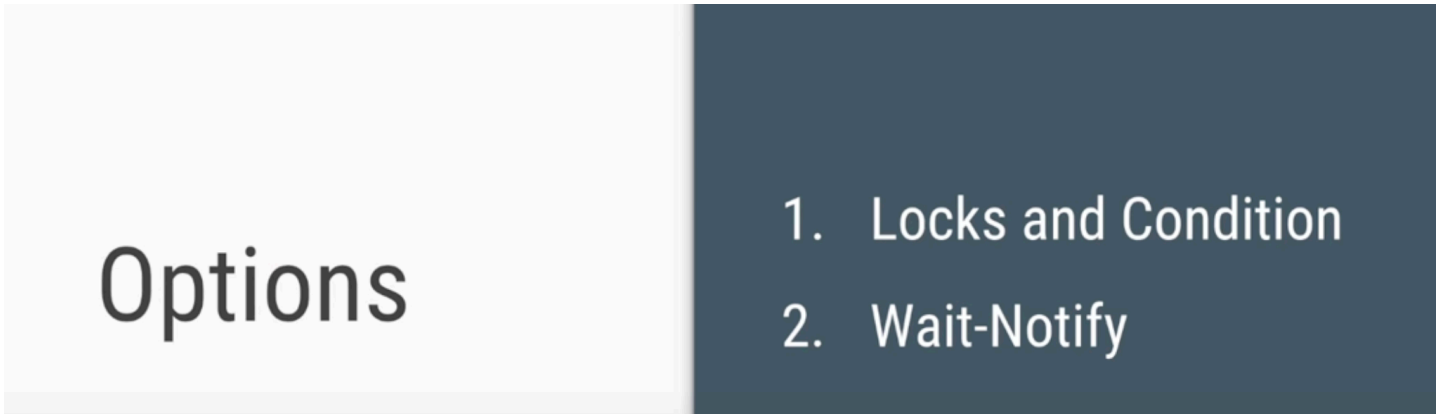
→ **Producer blocked if storage is full**



Solution using blocking queue:



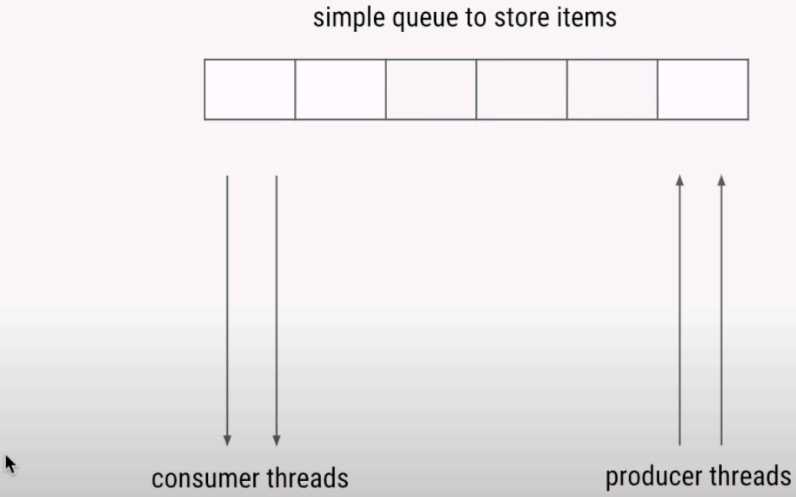
Without blocking queue:



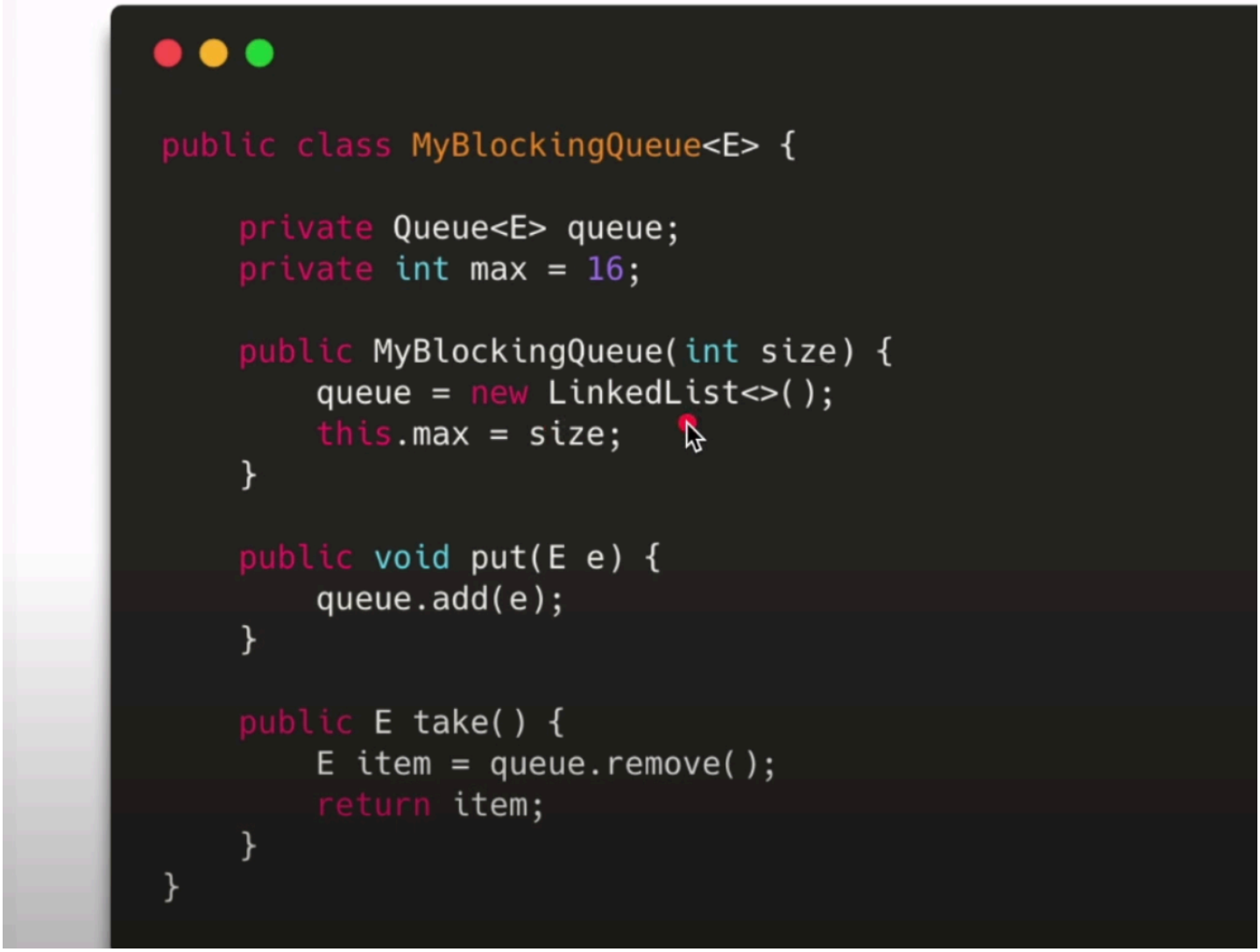
Implement using LOCK and condition:

Here need to create blocking queue by ourself

Basic skeleton



Code for skeleton



```
public class MyBlockingQueue<E> {

    private Queue<E> queue;
    private int max = 16;

    public MyBlockingQueue(int size) {
        queue = new LinkedList<>();
        this.max = size;
    }

    public void put(E e) {
        queue.add(e);
    }

    public E take() {
        E item = queue.remove();
        return item;
    }
}
```

Now Adding locks for the thread safety:

Code for skeleton

```
public class MyBlockingQueue<E> {

    private Queue<E> queue;
    private int max = 16;
    private ReentrantLock lock = new ReentrantLock(true);

    public MyBlockingQueue(int size) {
        queue = new LinkedList<>();
        this.max = size;
    }

    public void put(E e) {
        lock.lock();
        try {
            queue.add(e);
        } finally {
            lock.unlock();
        }
    }

    public E take() {
        lock.lock();
        try {
            E item = queue.remove();
            return item;
        } finally {
            lock.unlock();
        }
    }
```

can also use array, but need to track head and tail indexes

protected by lock

protected by same lock

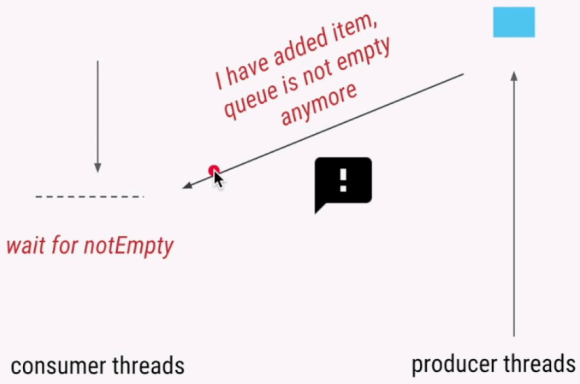
Use conditions to wait

```
private Condition notEmpty = lock.newCondition();
private Condition notFull = lock.newCondition();
```

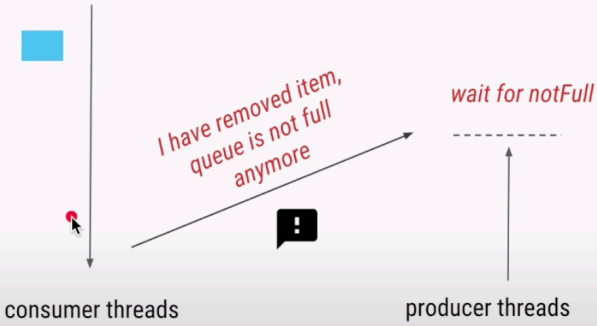
```
public void put(E e) {
    lock.lock();
    try {
        if (queue.size() == max) {
            notFull.await();
        }
        queue.add(e);
    } finally {
        lock.unlock();
    }
}
```

```
public E take() {
    lock.lock();
    try {
        if (queue.size() == 0) {
            notEmpty.await();
        }
        E item = queue.remove();
        return item;
    } finally {
        lock.unlock();
    }
}
```

Use conditions to signal



Use conditions to signal



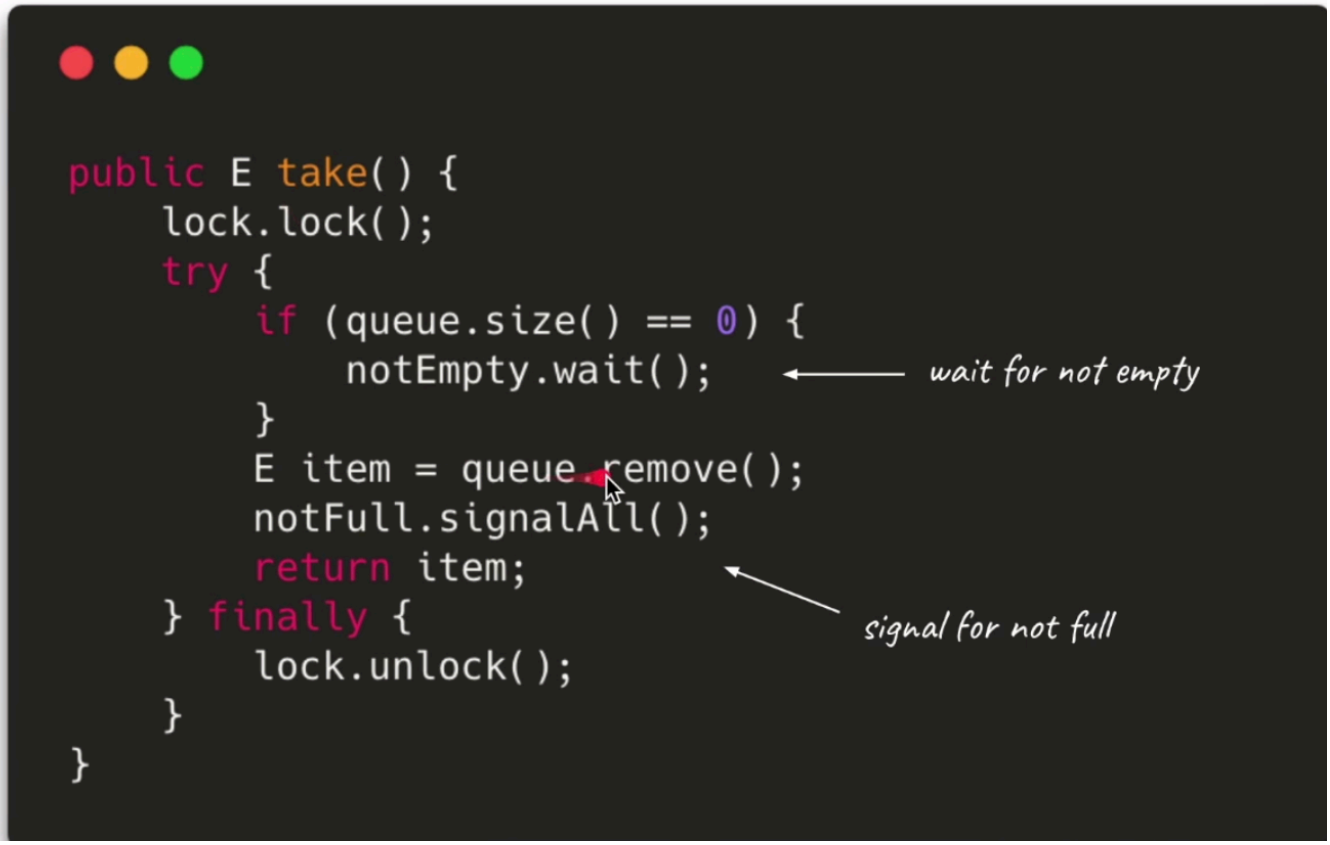
Use conditions to signal

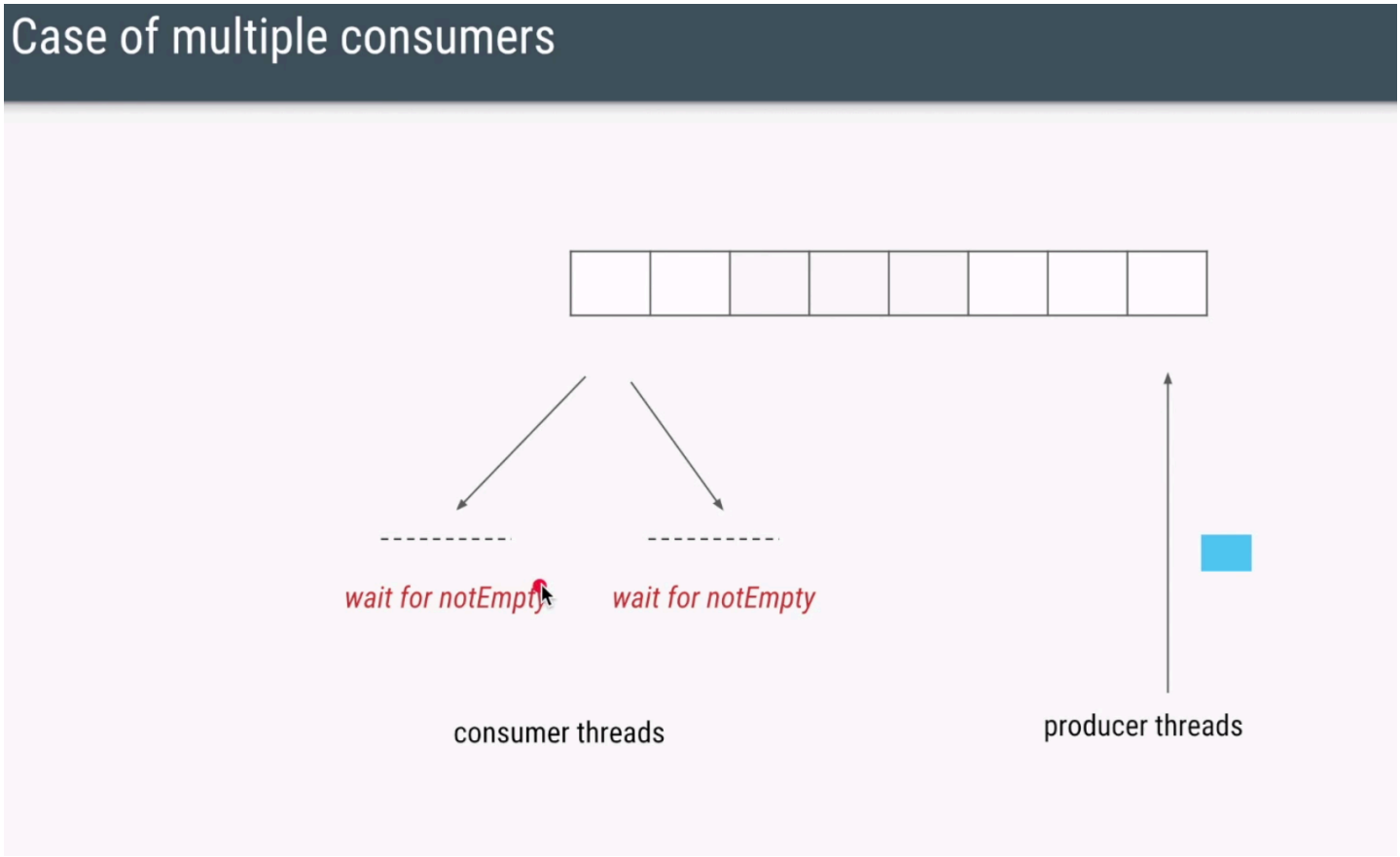
```
public void put(E e) {
    lock.lock();
    try {
        if (queue.size() == max) {
            notFull.wait();
        }
        queue.add(e);
        notEmpty.signalAll();
    } finally {
        lock.unlock();
    }
}
```

wait for not full

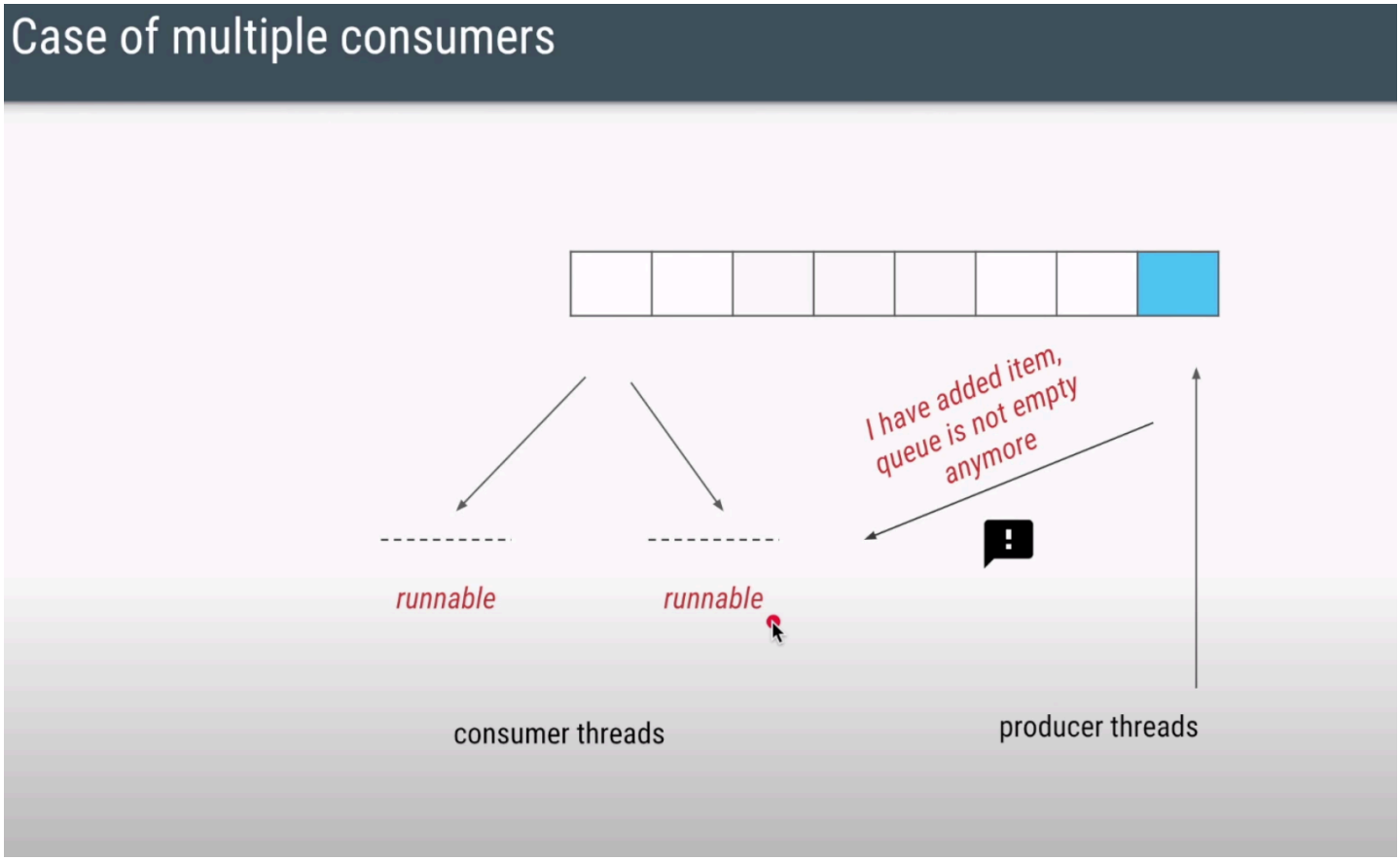
signal for not empty

Use conditions to signal

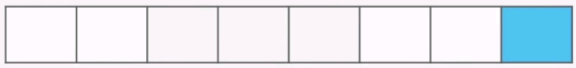


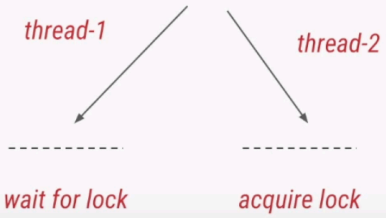


Prob : lets say queue is empty and multiple consumers are trying to get item:



Case of multiple consumers





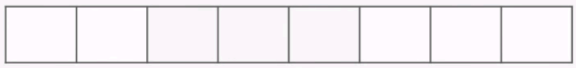
consumer threads

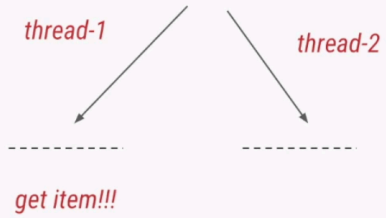
```
public E take() {
    lock.lock();
    try {
        if (queue.size() == 0) {
            notEmpty.await();
        }
        E item = queue.remove();
        notFull.signalAll();
        return item;
    } finally {
        lock.unlock();
    }
```

thread-1

thread-2

Case of multiple consumers





consumer threads

```
public E take() {
    lock.lock();
    try {
        while (queue.size() == 0) {
            notEmpty.await();
        }
        E item = queue.remove();
        notFull.signalAll();
        return item;
    } finally {
        lock.unlock();
    }
}
```

recheck queue size

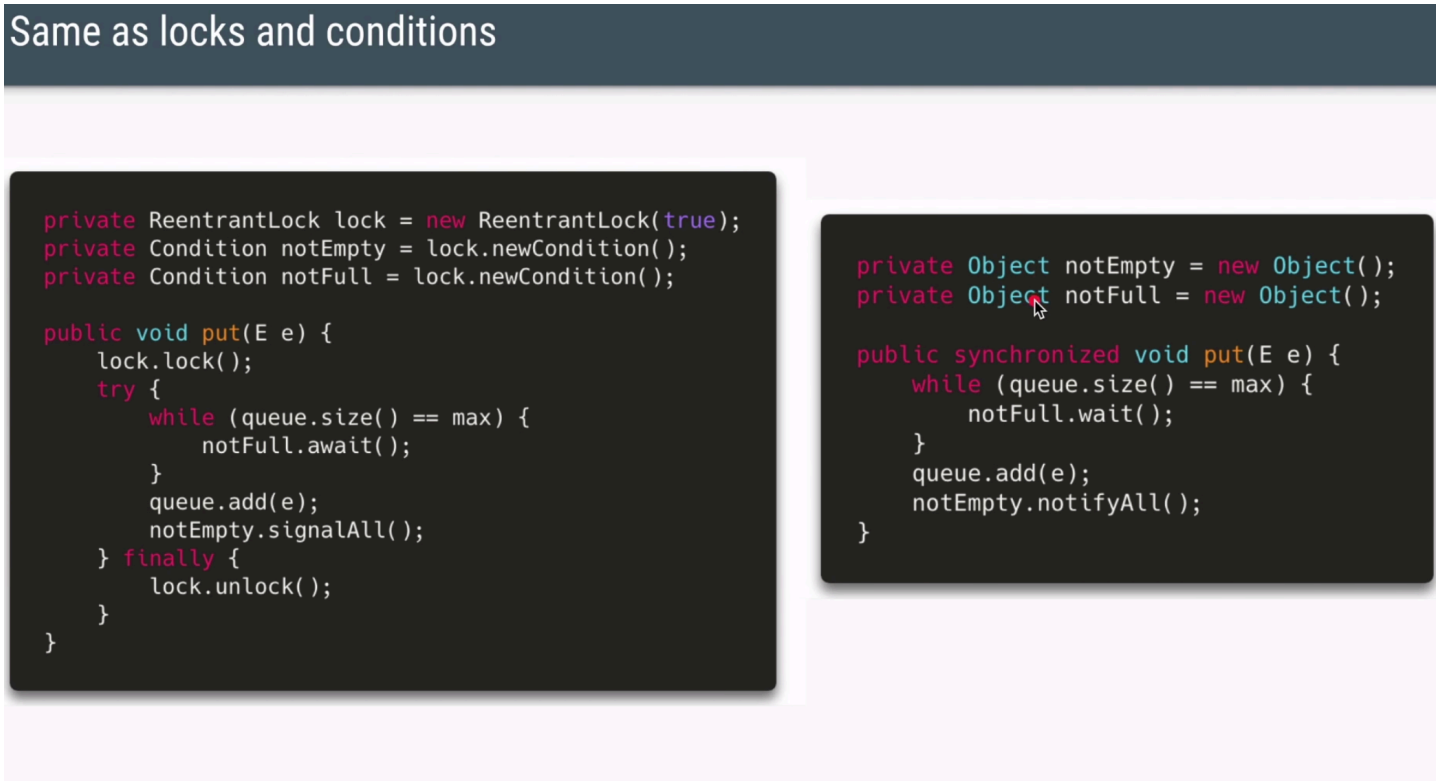
Full code

```
private int max;
private Queue<E> queue = new LinkedList<>();
private ReentrantLock lock = new ReentrantLock(true);
private Condition notEmpty = lock.newCondition();
private Condition notFull = lock.newCondition();

public MyBlockingQueue(int size) {
    queue = new LinkedList<>();
    this.max = size;
}

public void put(E e) {
    lock.lock();
    try {
        if (queue.size() == max) {
            notFull.wait();
        }
        queue.add(e);
        notEmpty.signalAll();
    } finally {
        lock.unlock();
    }
}

public E take() {
    lock.lock();
    try {
        if (queue.size() == 0) {
            notEmpty.wait();
        }
        E item = queue.remove();
        notFull.signalAll();
        return item;
    } finally {
        lock.unlock();
    }
}
```

Implementing using wait and notify:

Replace

1. lock→ Synchronized block
2. SignalAll-. NotifyAll
3. Condition → Object



How to timeout a thread?

- Stop a particular thread/task
- Timeout Condition

We want to stop a particular thread after 10 mins

We don't have any direct method to stop a thread

Does ThreadPool shutdown kill the threads?

shutdown()	1. No new tasks accepted. 2. Previously submitted tasks are executed.
shutdownNow()	1. No new tasks accepted. 2. Previously submitted tasks waiting in the queue are returned. 3. Tasks being run by the thread(s) are *attempted* to stop

No guarantees

Java threads cannot be killed.

They are cooperative.

You need to ask politely...

How to ask politely?

1. Interrupts
2. Volatile / AtomicBoolean

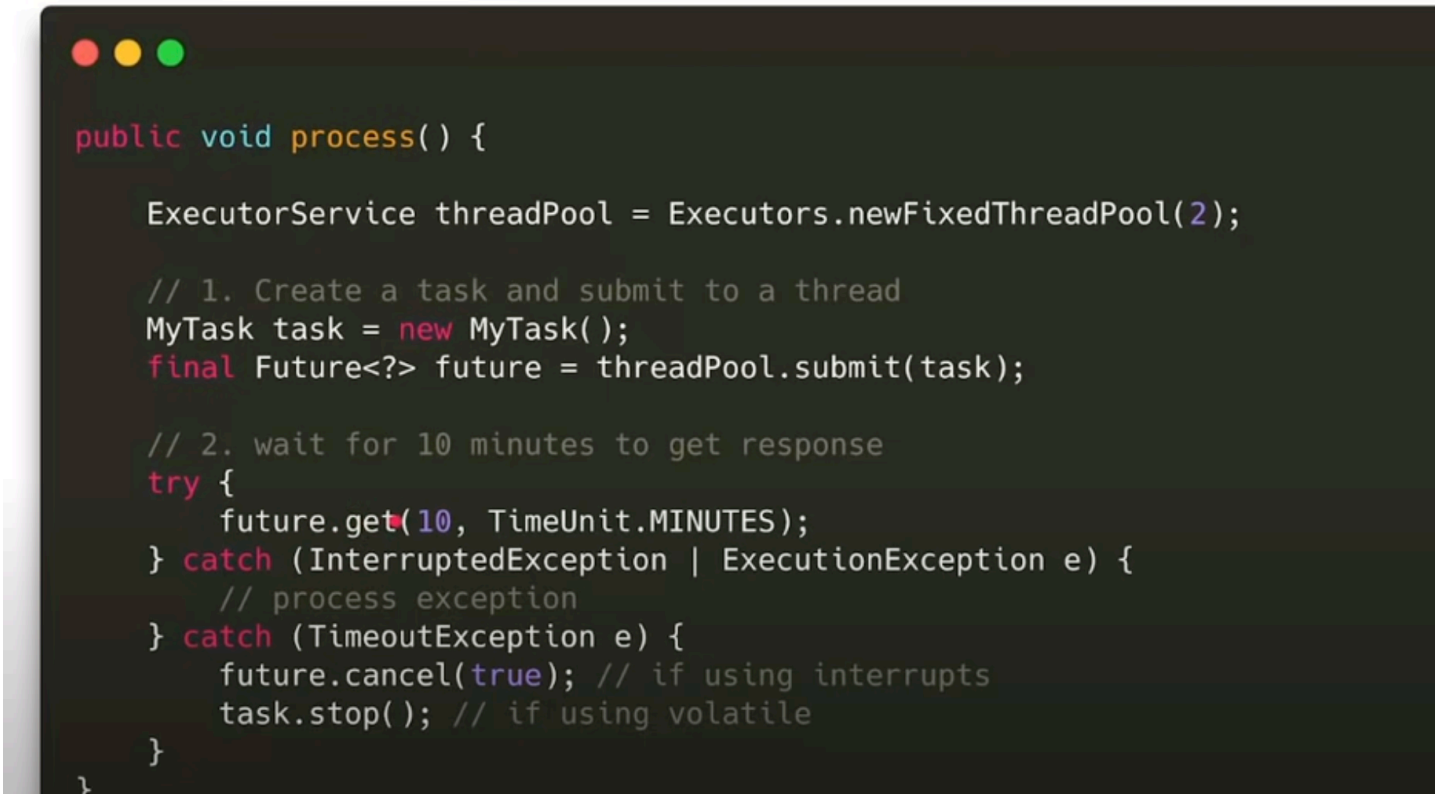
```java
public static void main(String[] args) {

    ExecutorService threadPool = Executors.newFixedThreadPool(2);

    threadPool.submit(() -> {
        while (!Thread.currentThread().isInterrupted()) {
            // next step
        }
    });

    // 2. TODO: timeout for 10 minutes

    // 3. stop the thread
    threadPool.shutdownNow();
}
```

This internally calls thread.interrupt for all running threads.

```java
public void process() {

    // 1. Create a task and submit to a thread
    MyTask task = new MyTask();
    Thread t1 = new Thread(task);
    t1.start();

    // 2. TODO: timeout for 10 minutes

    // 3. ask task to stop using volatile
    task.keepRunning = false;
}

private class MyTask implements Runnable {

    public volatile boolean keepRunning = true;

    @Override
    public void run() {
        while (keepRunning == true) {
            // steps
        }
```

Same will work for ThreadPool

future.cancel with a timeout



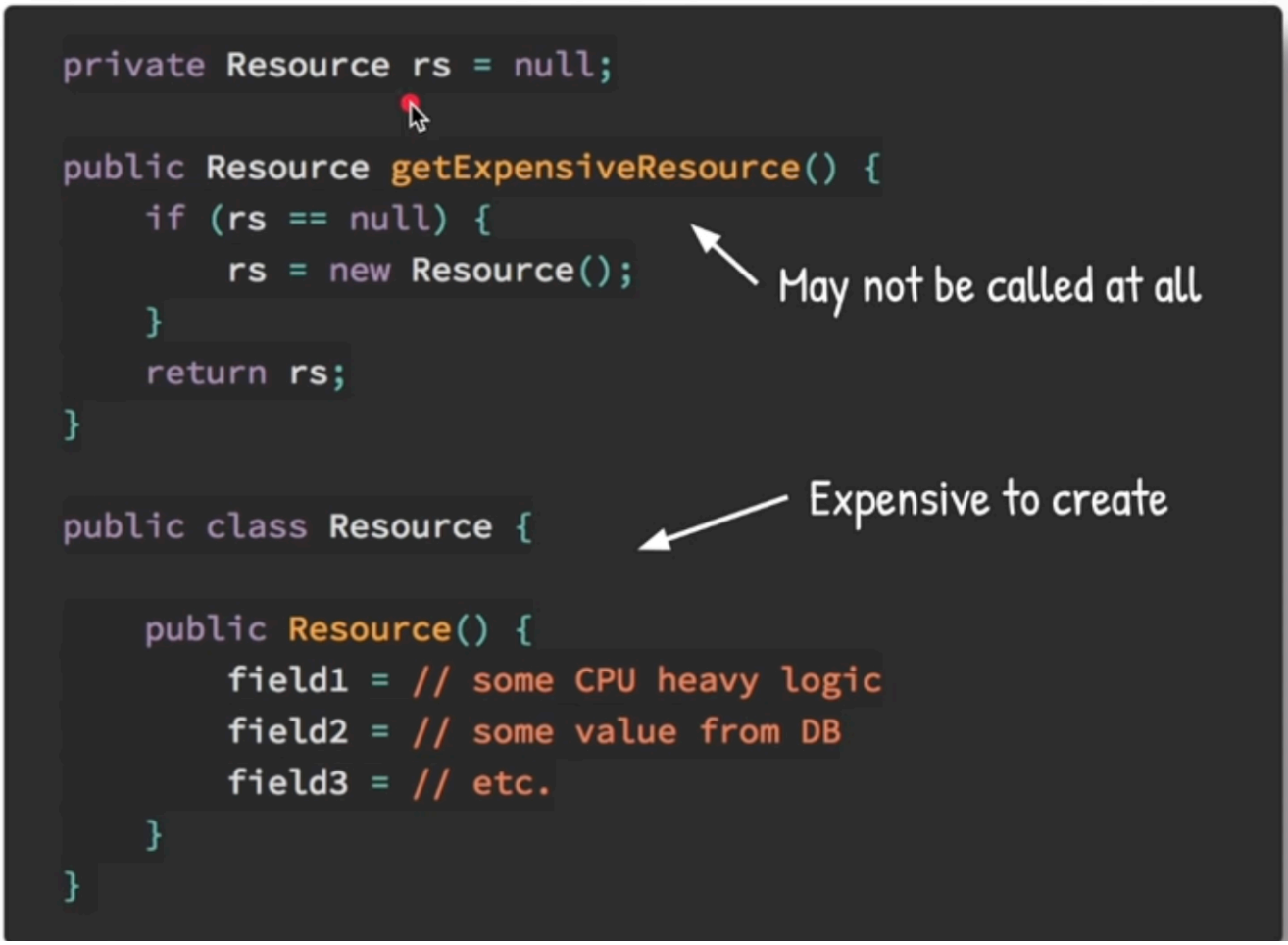
```
public void process() {

    ExecutorService threadPool = Executors.newFixedThreadPool(2);

    // 1. Create a task and submit to a thread
    MyTask task = new MyTask();
    final Future<?> future = threadPool.submit(task);

    // 2. wait for 10 minutes to get response
    try {
        future.get(10, TimeUnit.MINUTES);
    } catch (InterruptedException | ExecutionException e) {
        // process exception
    } catch (TimeoutException e) {
        future.cancel(true); // if using interrupts
        task.stop(); // if using volatile
    }
}
```

Singleton and Double Checked Locking:

Eager initialization

```
private Resource rs = new Resource();

public Resource getExpensiveResource(){
    return rs;
}

public Resource() {
    field1 = // some CPU heavy logic
    field2 = // some value from DB
    field3 = // etc.
}
```

Check once - Lazy initialization



Check once is unsafe

```
private Resource rs = null;

public Resource getExpensiveResource() {
    if (rs == null) {
        rs = new Resource();
    }
    return rs;
}
```

thread-1

thread-2

Lock and then check once

```
private Resource rs = null;

public Resource getExpensiveResource() {

    synchronized(this) {
        if (rs == null) {
            rs = new Resource();
        }
    }
    return rs;
}
```

Lock and then check once

```
private Resource rs = null;

public Resource getExpensiveResource() {

    synchronized(this) {
        if (rs == null) {
            rs = new Resource();
        }
    }
    return rs;
}
```

thread-2 (waiting)

thread-1

Every thread will have to synchronize!

```
private Resource rs = null;

public Resource getExpensiveResource() {

    synchronized(this) {
        if (rs == null) {
            rs = new Resource();
        }
    }
    return rs;
}
```

not efficient

Double Checked Locking

```
private Resource rs = null;

public Resource getExpensiveResource() {

    if (rs == null) {
        synchronized(this) {
            if (rs == null) {
                rs = new Resource();
            }
        }
    }
    return rs;
}
```

synchronize only if resource is not initialized

Double Checked Locking

```
private Resource rs = null;

public Resource getExpensiveResource() {

    if (rs == null) {                    ← check
        synchronized(this) {             ← lock
            if (rs == null) {            ← check
                rs = new Resource();
            }
        }
    }
    return rs;
}
```

Double Checked Locking

```
private Resource rs = null;

public Resource getExpensiveResource() {

    if (rs == null) {                    ← thread-2
        synchronized(this) {
            if (rs == null) {
                rs = new Resource();
            }
        }
    }
    return rs;
}
```

✓ construct empty resource()
✓ assign to rs
call constructor

```
public Resource() {
    field1 = // some CPU heavy logic
    field2 = // some value from DB
    field3 = // etc.
}
```

Double Checked Locking - Correct!

```
private volatile Resource rs = null;

public Resource getExpensiveResource() {

    if (rs == null) {
        synchronized(this) {
            if (rs == null) {
                rs = new Resource();
            }
        }
    }
    return rs;
}
```

Singleton

```
public class Resource {

    private static volatile Resource instance = null;

    private Resource() {
        // ...
    }

    public static Resource getInstance() {
```