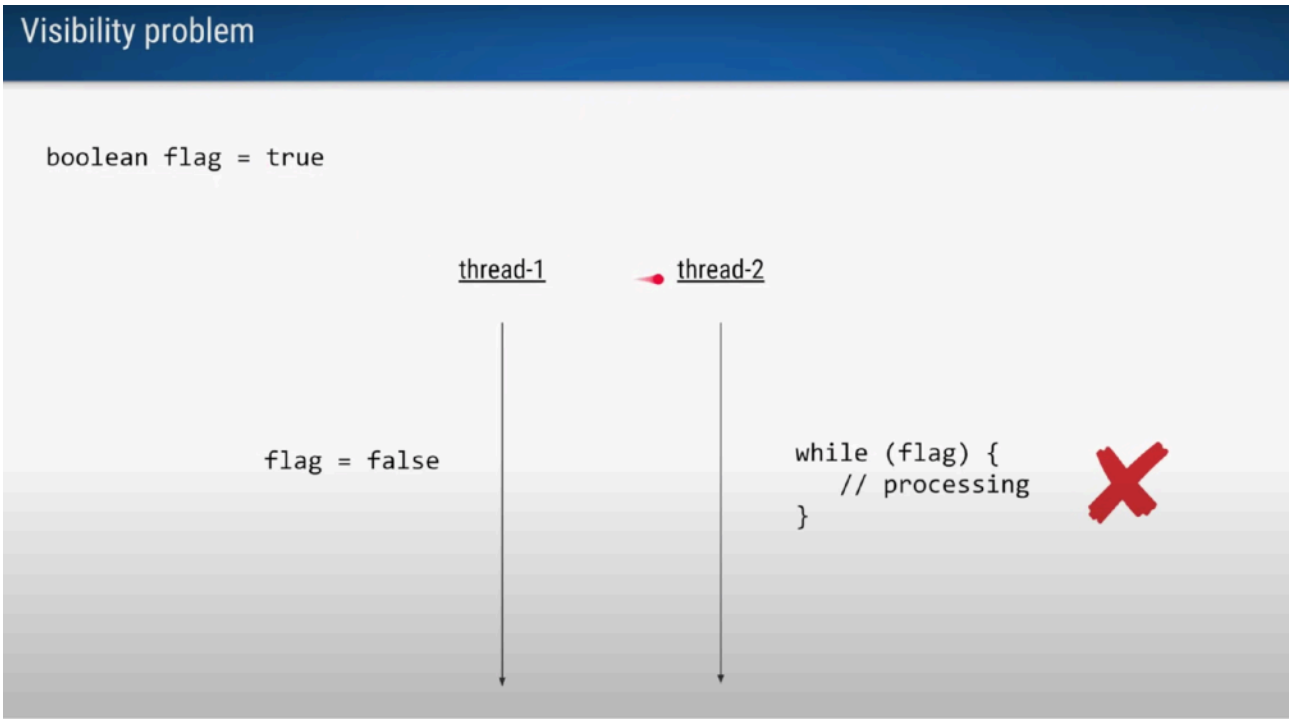
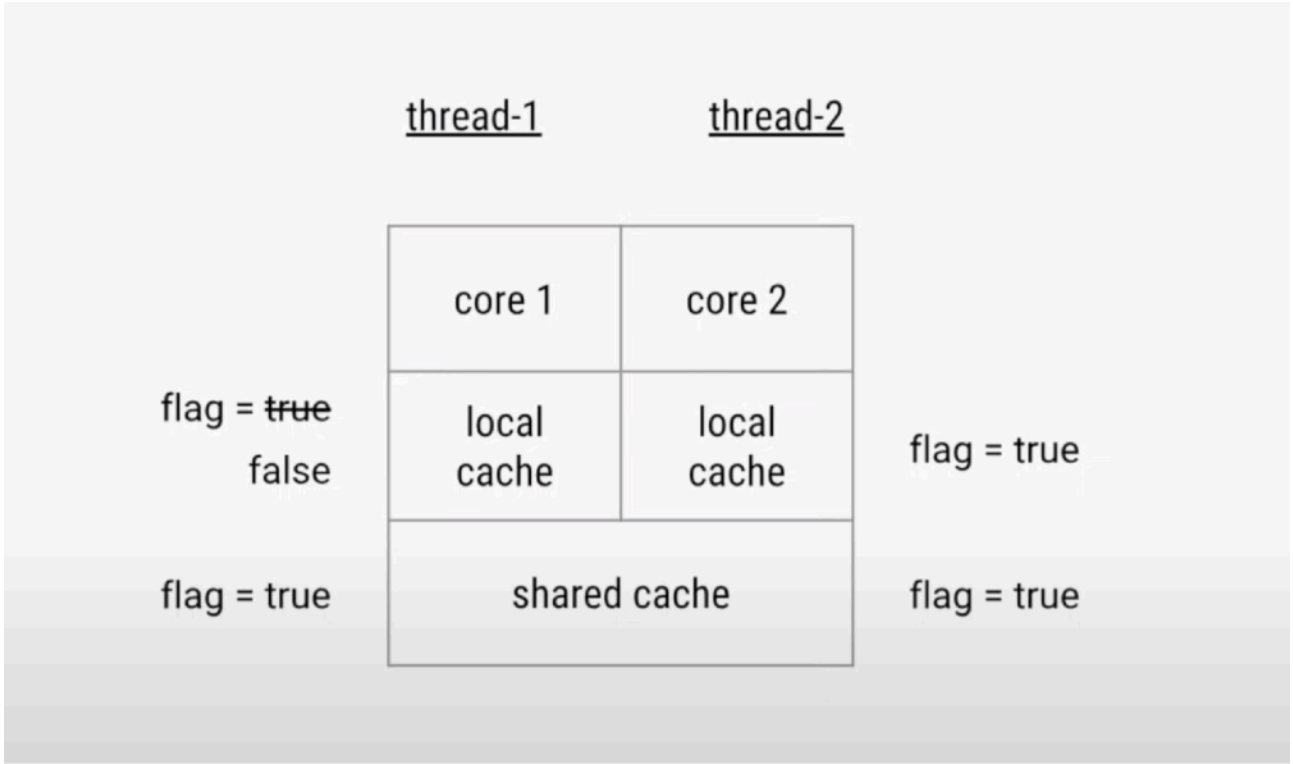
Visibility Problem :



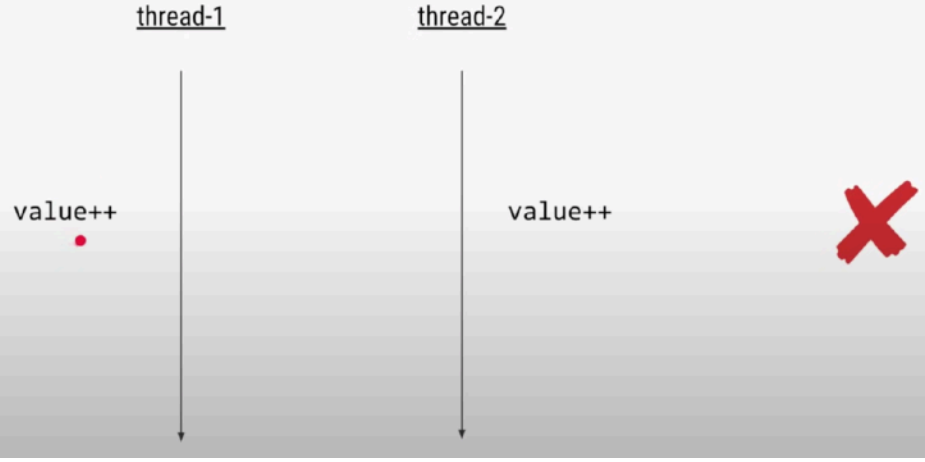


Visibility problem

```
volatile boolean flag = true
```

Synchronization problem

```
int value = 1;
```



Synchronization problem

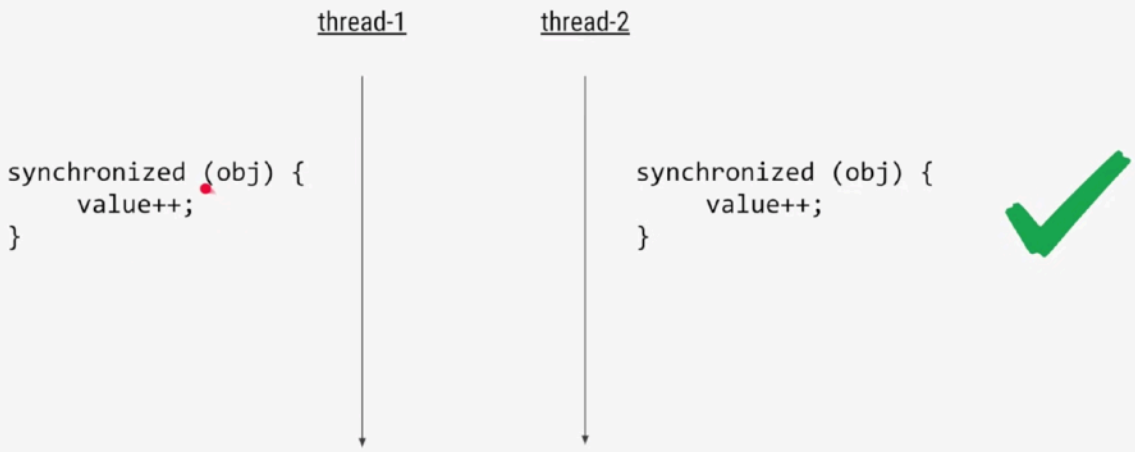
```
volatile int value = 1;
```

Even with volatile

#	Thread-1	Thread-2
1	Read value (=1)	
2		Read value (=1)
3	Add 1 and write (=2)	
4		Add 1 and write (=2)

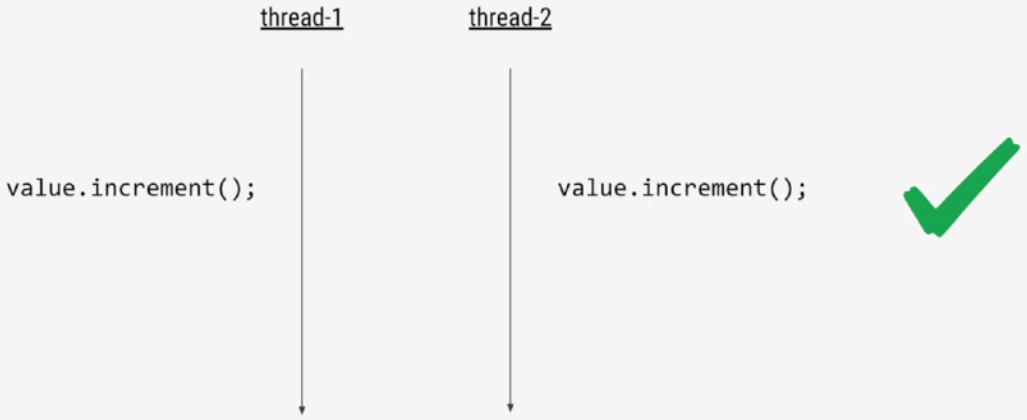
Synchronization solutions - #1

```
volatile int value = 1;
```



Synchronization solutions - #1

```
AtomicInteger value = new AtomicInteger(1);
```



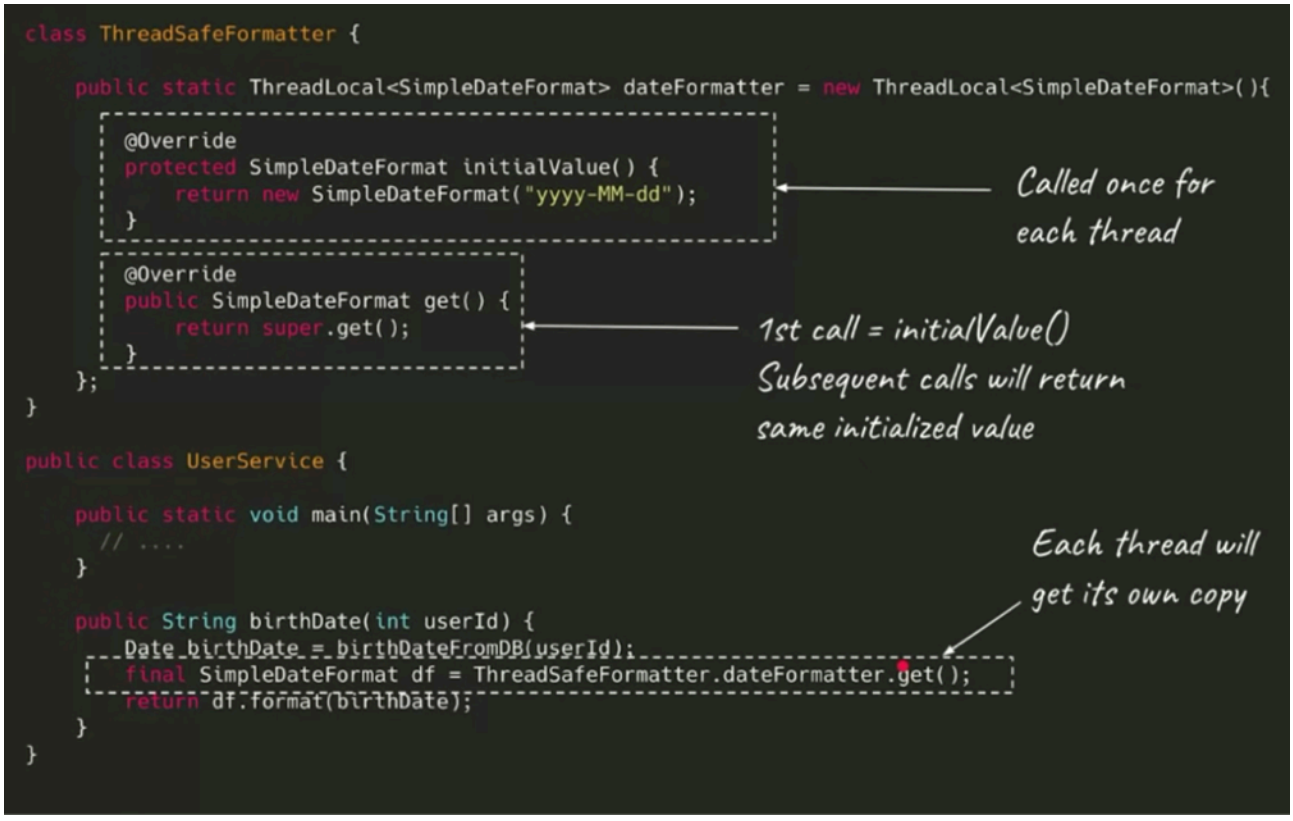
Many methods for various compound operations

- incrementAndGet
- decrementAndGet
- addAndGet (int delta)
- compareAndSet (int expectedValue, int newValue)

Typical Use Cases

Type	Use Case
volatile	Flags
AtomicInteger AtomicLong	Counters
AtomicReference	Caches (building new cache in background and replacing atomically) Used by some internal classes Non-blocking algorithms

Thread Local:



Async Programming :

Synchronous API

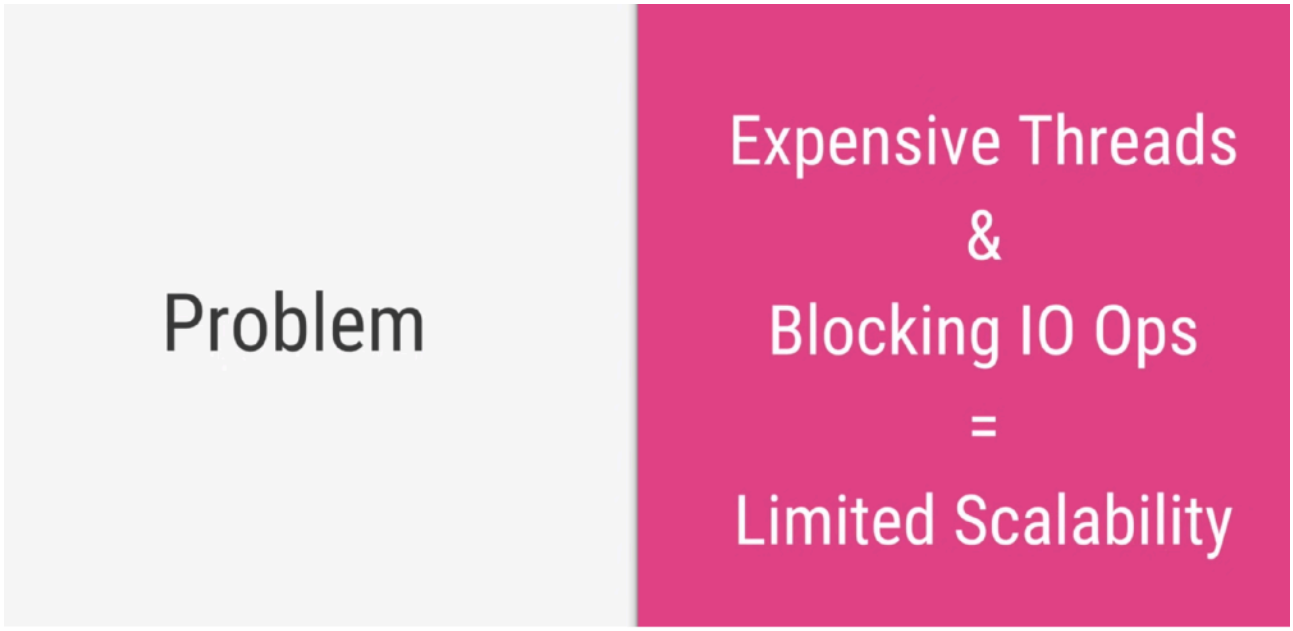
```
for (Integer id : employeeIds) {

    // Step 1: Fetch Employee details from DB
    Future<Employee> future = service.submit(new EmployeeFetcher(id));
    Employee emp = future.get(); // blocking

    // Step 2: Fetch Employee tax rate from REST service
    Future<TaxRate> rateFuture = service.submit(new TaxRateFetcher(emp));
    TaxRate taxRate = rateFuture.get(); // blocking

    // Step 3: Calculate current year tax
    BigDecimal tax = calculateTax(emp, taxRate);

    // Step 4: Send email to employee using REST service
    service.submit(new SendEmail(emp, tax));
}
```



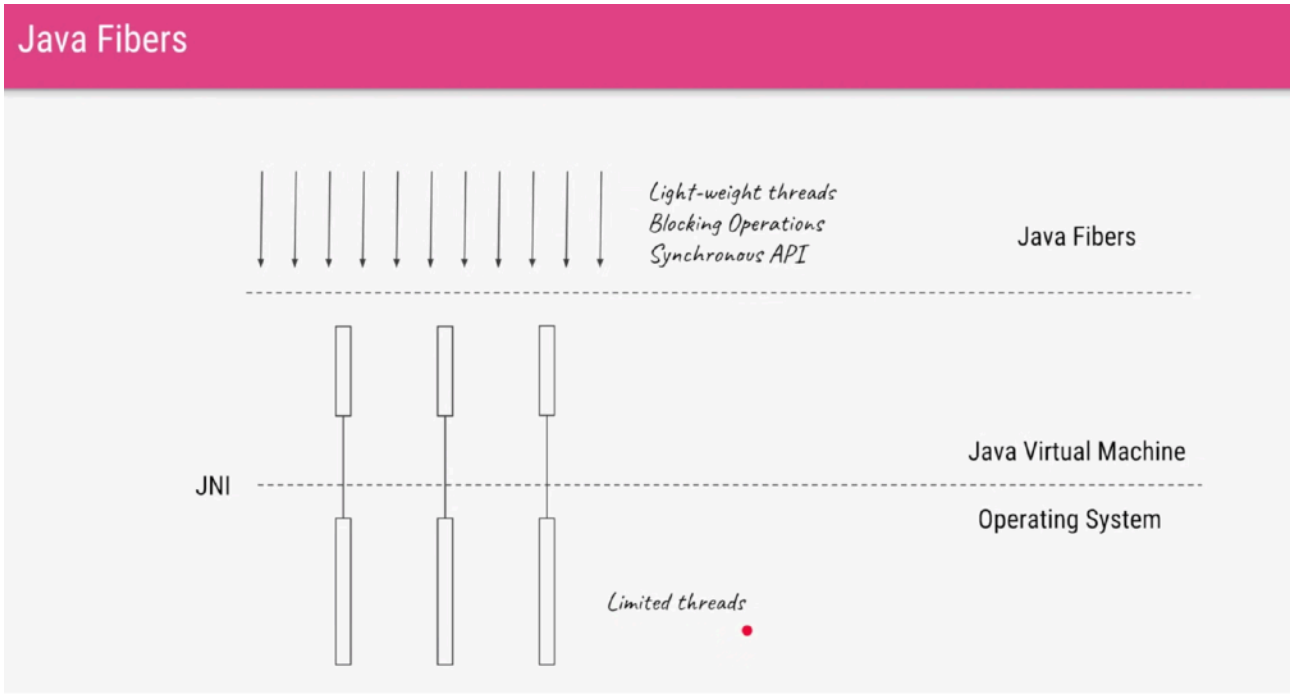
Solution

Non-blocking IO
&
Asynchronous API

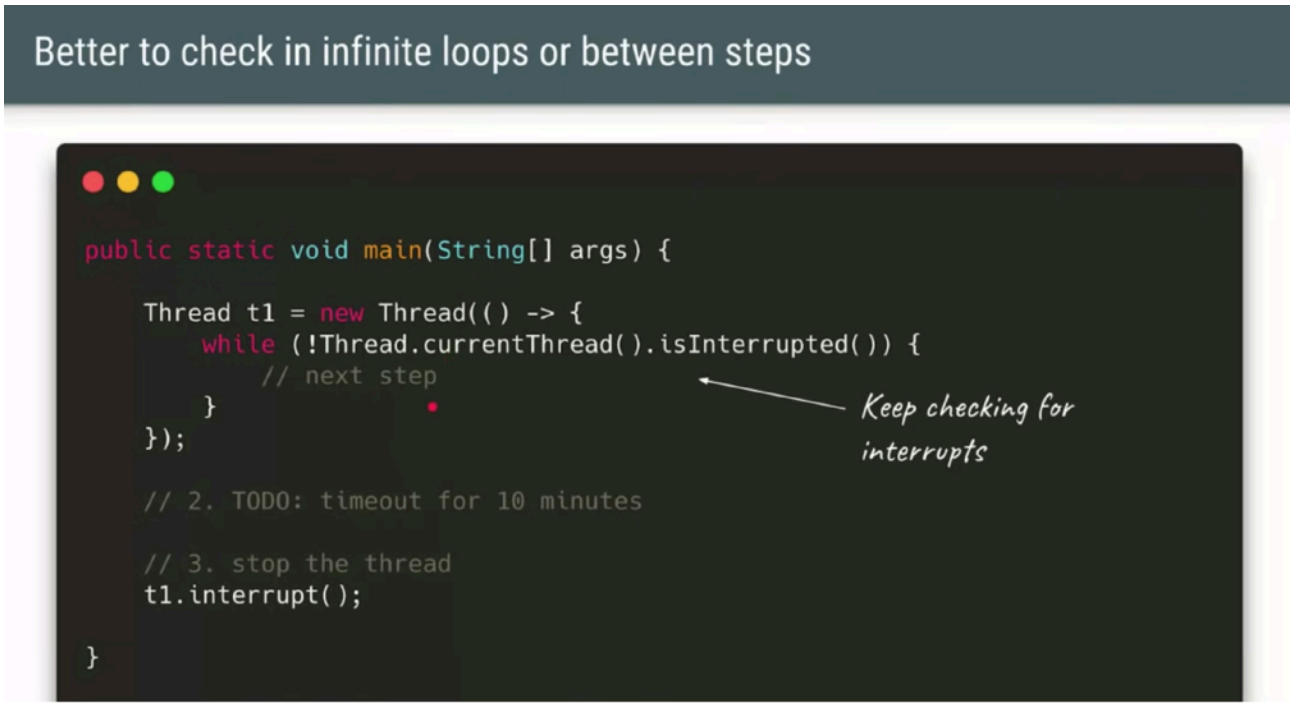
Asynchronous API - Callbacks

```
for (Integer id : employeeIds) {
    CompletableFuture.supplyAsync(() -> fetchEmployee(id))
            .thenApplyAsync(employee -> fetchTaxRate(employee))
            .thenApplyAsync(taxRate -> calculateTax(taxRate))
            .thenAcceptAsync(taxValue -> sendEmail(taxValue));
}
```

Callback chaining (similar to JS)



A task is running separate thread. Stop the task if it exceed 10 minutes :



Interrupt: Works with Futures and Callables as well

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

    ExecutorService threadPool = Executors.newFixedThreadPool(2);

    Future<?> future = threadPool.submit(() -> {
        while (!Thread.currentThread().isInterrupted()) {
            // next step
        }
    });

    // 2. TODO: timeout for 10 minutes

    // 3. stop the thread
    future.cancel(true);
}
```

Calls thread.interrupt for thread running the task

Volatile

```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

AtomicBoolean

```
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. stop the thread
    task.stop();
}

private class MyTask implements Runnable {

    public AtomicBoolean keepRunning = new AtomicBoolean(true);

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

    public void stop() {
        keepRunning.set(false);
    }
}
```

Producer Consumer :


public static void main(String[] args) {
BlockingQueue<Item> queue = new ArrayBlockingQueue<>(10);
Handles concurrent thread access
// Producer
final Runnable producer = () -> {
while (true) {
queue.put(createItem());
}
};
thread blocks if queue full
new Thread(producer).start();
new Thread(producer).start();
// Consumer
final Runnable consumer = () -> {
while (true) {
Item i = queue.take();
process(i);
}
};
thread blocks if queue empty
new Thread(consumer).start();
new Thread(consumer).start();
Thread.sleep(1000);
}
try-catch for take and put skipped for brevity

With Lock:

```
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();
    }
}
```

```java
private ReentrantLock lock = new ReentrantLock(true);
private Condition notEmpty = lock.newCondition();
private Condition notFull = lock.newCondition();

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

```java
private Object notEmpty = new Object();
private Object notFull = new Object();

public synchronized E take() {
    if (queue.size() == 0) {
        notEmpty.wait();
    }
    E item = queue.remove();
    notFull.notifyAll();
    return item;
}
```

Visibility problem

```
boolean flag = true
```