CS 10: Problem solving via Object Oriented Programming

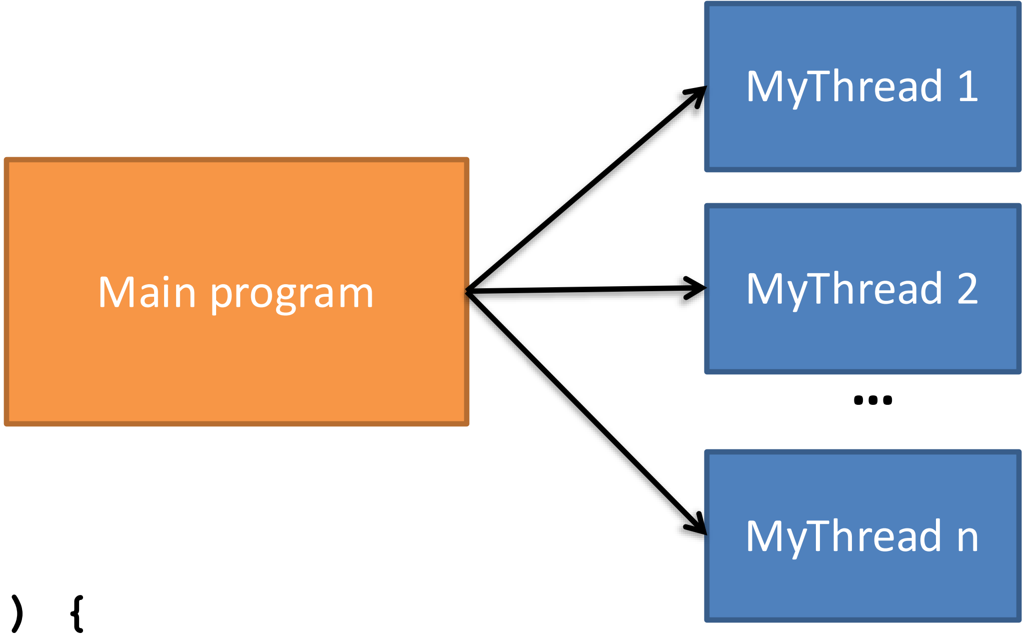
Synchronization

Agenda

1. Threads and interleaving execution
2. Producer/consumer
3. Deadlock, starvation

Threads are a way for multiple processes to run "concurrently"

Threads



```
main() {
MyThread t = new MyThread();

//start thread at run method, main
thread keeps running
t.start()

//halt main until thread finishes
t.join()
```

Assume `MyThread` is a class that extends `Thread`

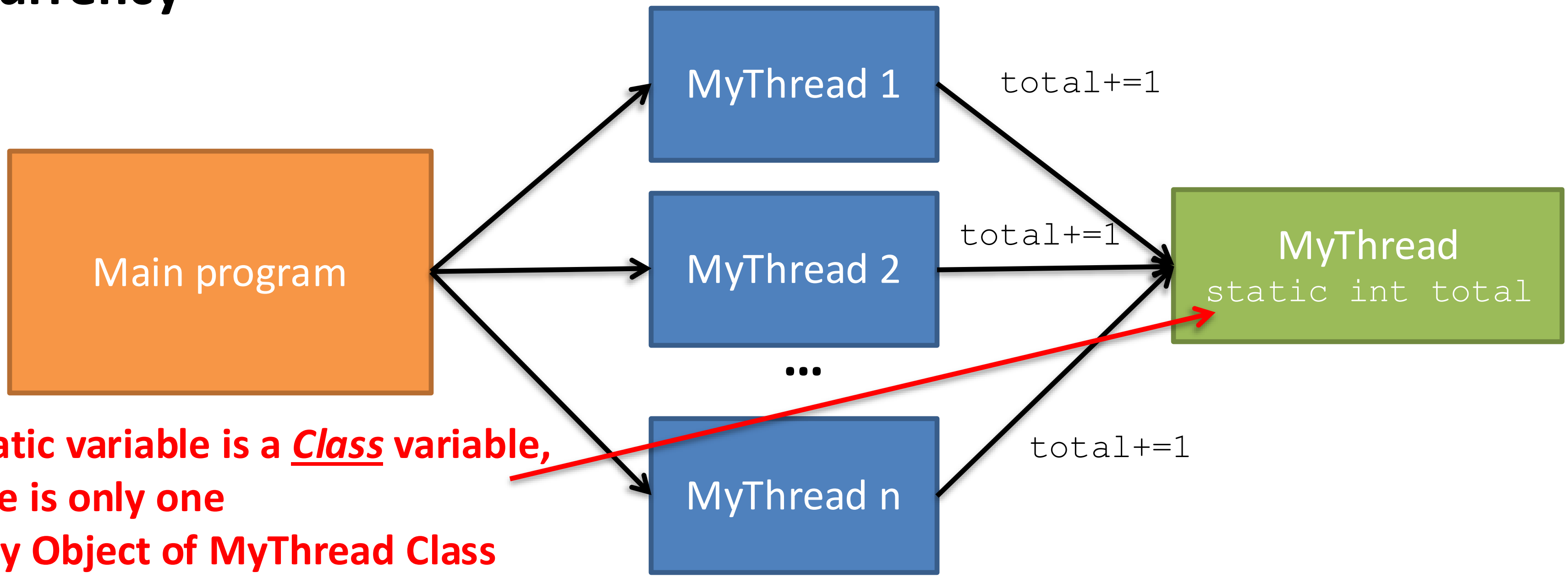
`MyThread` must implement a `run` method

Execution begins by calling `start` on a `MyThread` object, `run` method then executes

Can call `join` to halt main program until thread finishes

Concurrent threads can access the same resources; this can cause problems

Concurrency

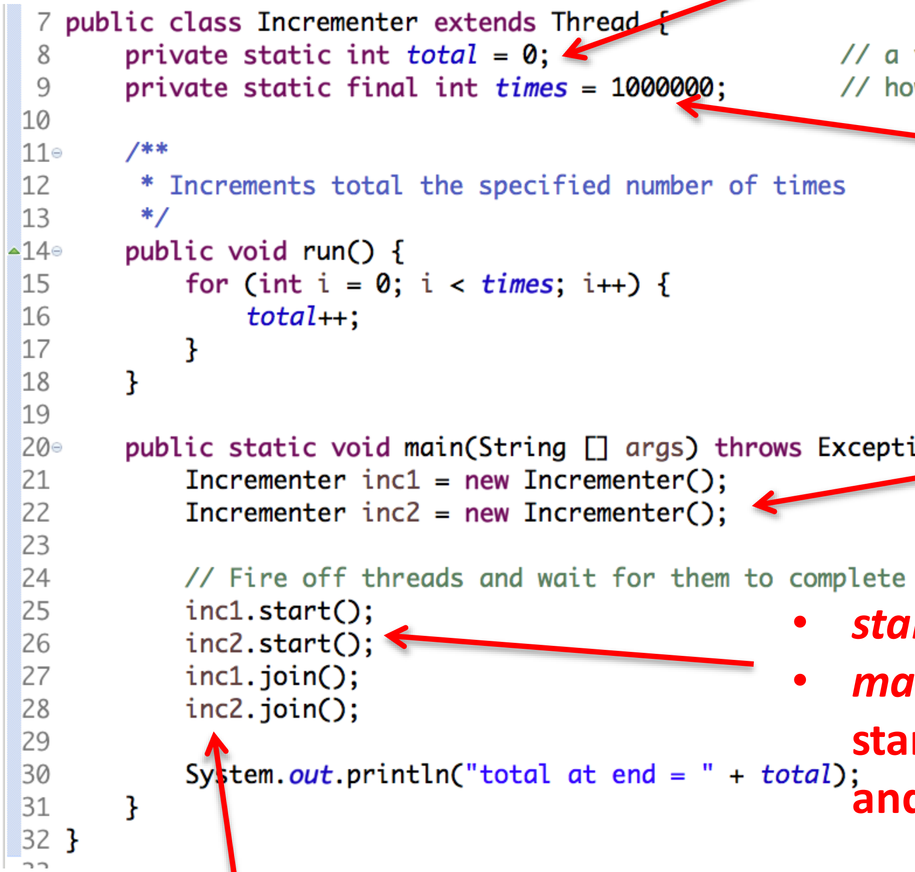


- **A static variable is a *Class* variable, there is only one**
- **Every Object of MyThread Class references the *same* static variable**

- Threads can be interrupted at any time by the Operating System and another Thread may run
- When each Thread tries to increment `total`, it gets a current copy of `total`, adds 1, then stores it back in memory
- What can go wrong?

Let’s make it interesting, what is the final value of total?

Incrementer.java



```
public class Incrementer extends Thread {
    private static int total = 0;               // a variable shared by all incrementers
    private static final int times = 1000000;   // how many times to increment total, in each thread

    /**
     * Increments total the specified number of times
     */
    public void run() {
        for (int i = 0; i < times; i++) {
            total++;
        }
    }

    public static void main(String [] args) throws Exception {
        Incrementer inc1 = new Incrementer();
        Incrementer inc2 = new Incrementer();

        // Fire off threads and wait for them to complete
        inc1.start();
        inc2.start();
        inc1.join();
        inc2.join();

        System.out.println("total at end = " + total);
    }
}
```

total **is static so it is a Class variable (one** ***total*** **for all Incrementer Objects)**

One million (not a trick)

What will ***total*** **be at end?**
Top three guesses?

Two Incrementer Objects that extend Thread (so must implement ***run()*** **method)**

- ***start()*** **begins Thread running and calls** ***run()*** **method**
- ***main()*** **continues running after** ***inc1.start(), so inc2*** **starts immediately after** ***inc1*** **(*main()* does not block and wait for inc1 to finish)**

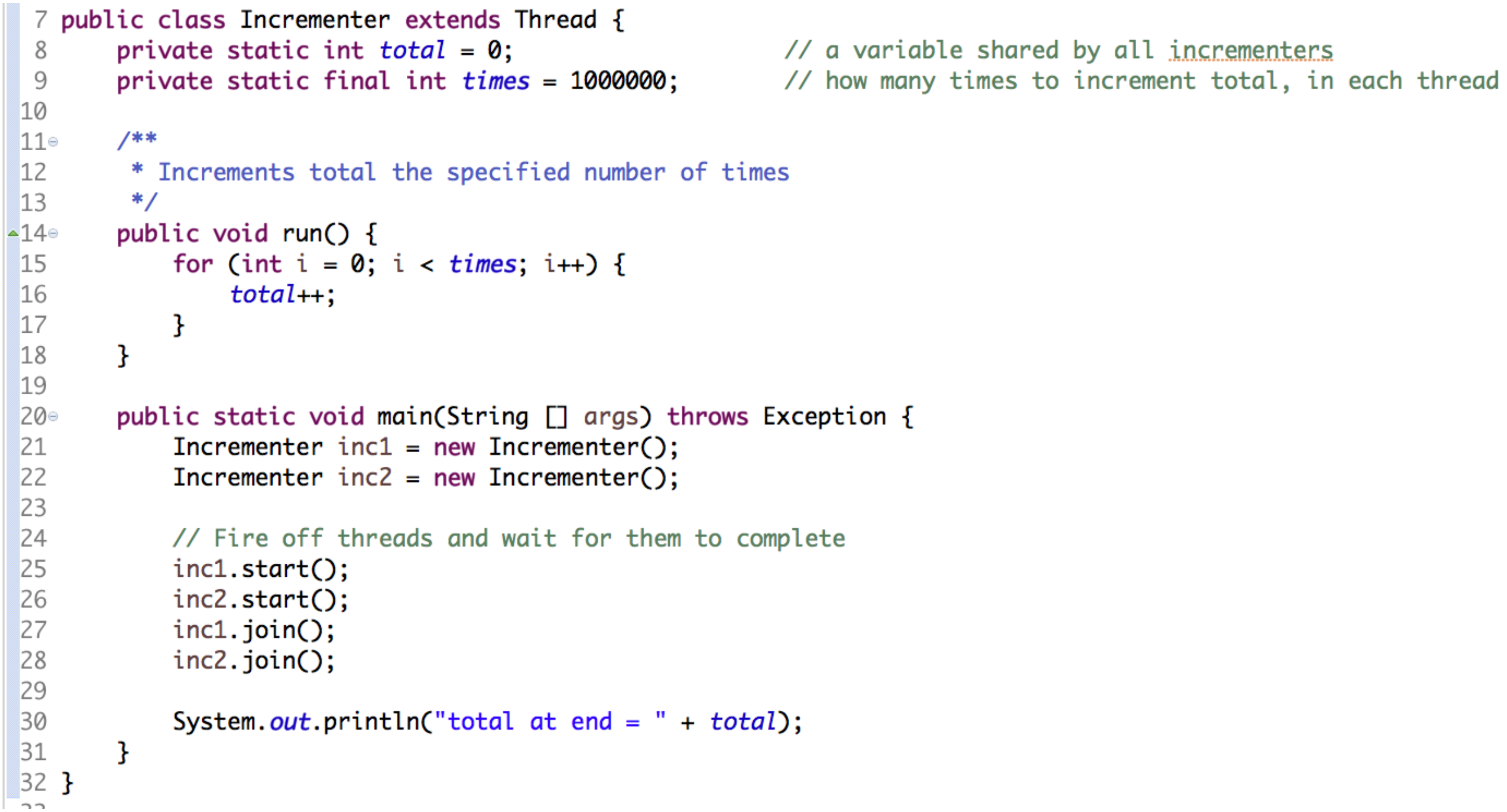
- ***inc1.join()*** **causes** ***main()*** **to block until** ***inc1.run()*** **finishes**
- ***inc2.join()*** **causes** ***main()*** **to block until** ***inc2.run()*** **finishes**

Move to next slide only after running Incrementer.java

Run Incrementer.java before proceeding

Threads can be interrupted at any point, this can cause unexpected behavior

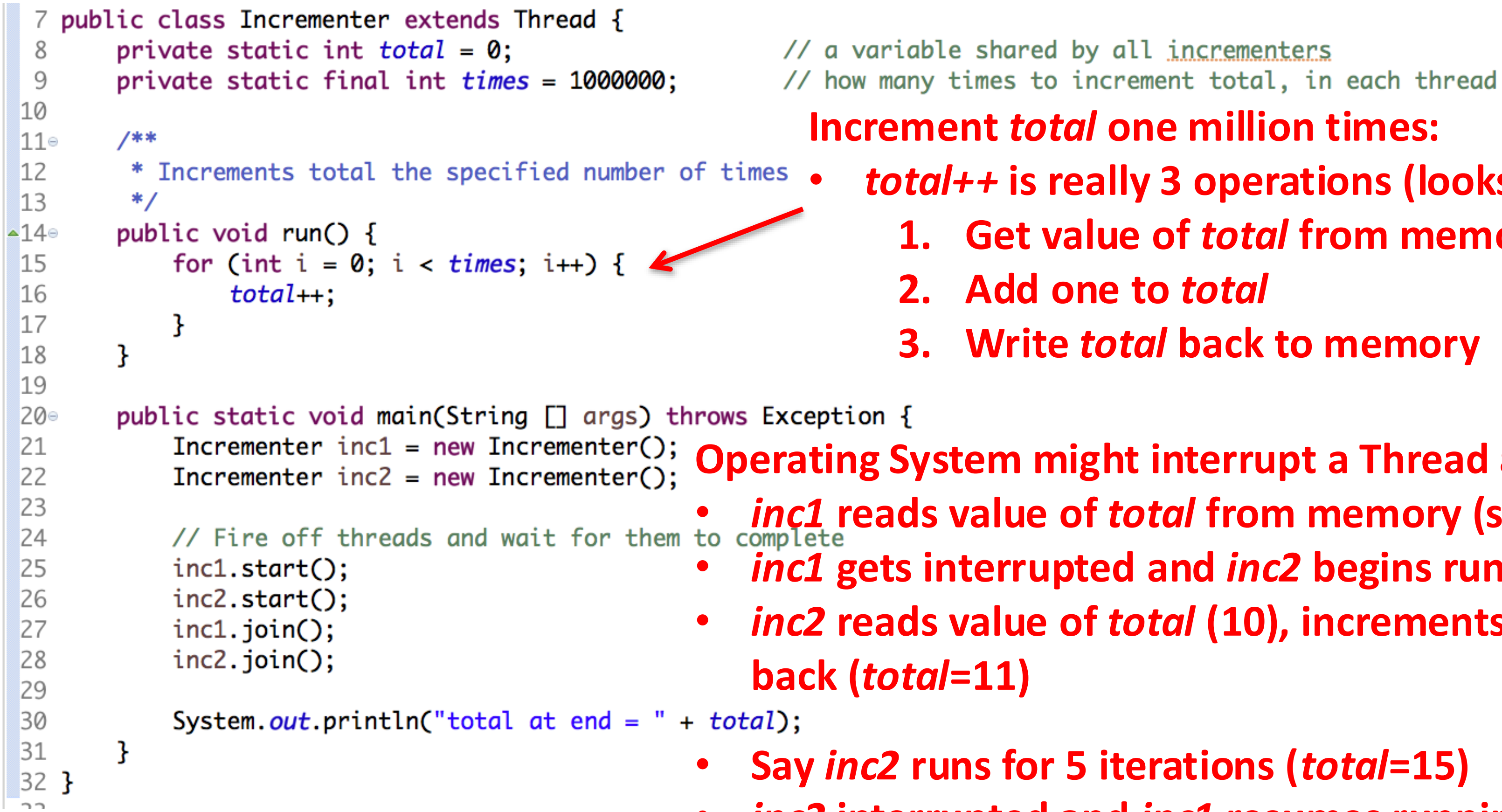
Incrementer.java



```java
public class Incrementer extends Thread {
    private static int total = 0;                   // a variable shared by all incrementers
    private static final int times = 1000000;       // how many times to increment total, in each thread

    /**
     * Increments total the specified number of times
     */
    public void run() {
        for (int i = 0; i < times; i++) {
            total++;
        }
    }

    public static void main(String [] args) throws Exception {
        Incrementer inc1 = new Incrementer();
        Incrementer inc2 = new Incrementer();

        // Fire off threads and wait for them to complete
        inc1.start();
        inc2.start();
        inc1.join();
        inc2.join();

        System.out.println("total at end = " + total);
    }
}
```

Threads can be interrupted at any point, this can cause unexpected behavior

Incrementer.java



```java
public class Incrementer extends Thread {
    private static int total = 0;                   // a variable shared by all incrementers
    private static final int times = 1000000;       // how many times to increment total, in each thread

    /**
     * Increments total the specified number of times
     */
    public void run() {
        for (int i = 0; i < times; i++) {
            total++;
        }
    }

    public static void main(String [] args) throws Exception {
        Incrementer inc1 = new Incrementer();
        Incrementer inc2 = new Incrementer();

        // Fire off threads and wait for them to complete
        inc1.start();
        inc2.start();
        inc1.join();
        inc2.join();

        System.out.println("total at end = " + total);
    }
}
```

Increment *total* one million times:

- ***total++*** **is really 3 operations (looks like 1)**
  1. **Get value of *total* from memory**
  2. **Add one to *total***
  3. **Write *total* back to memory**

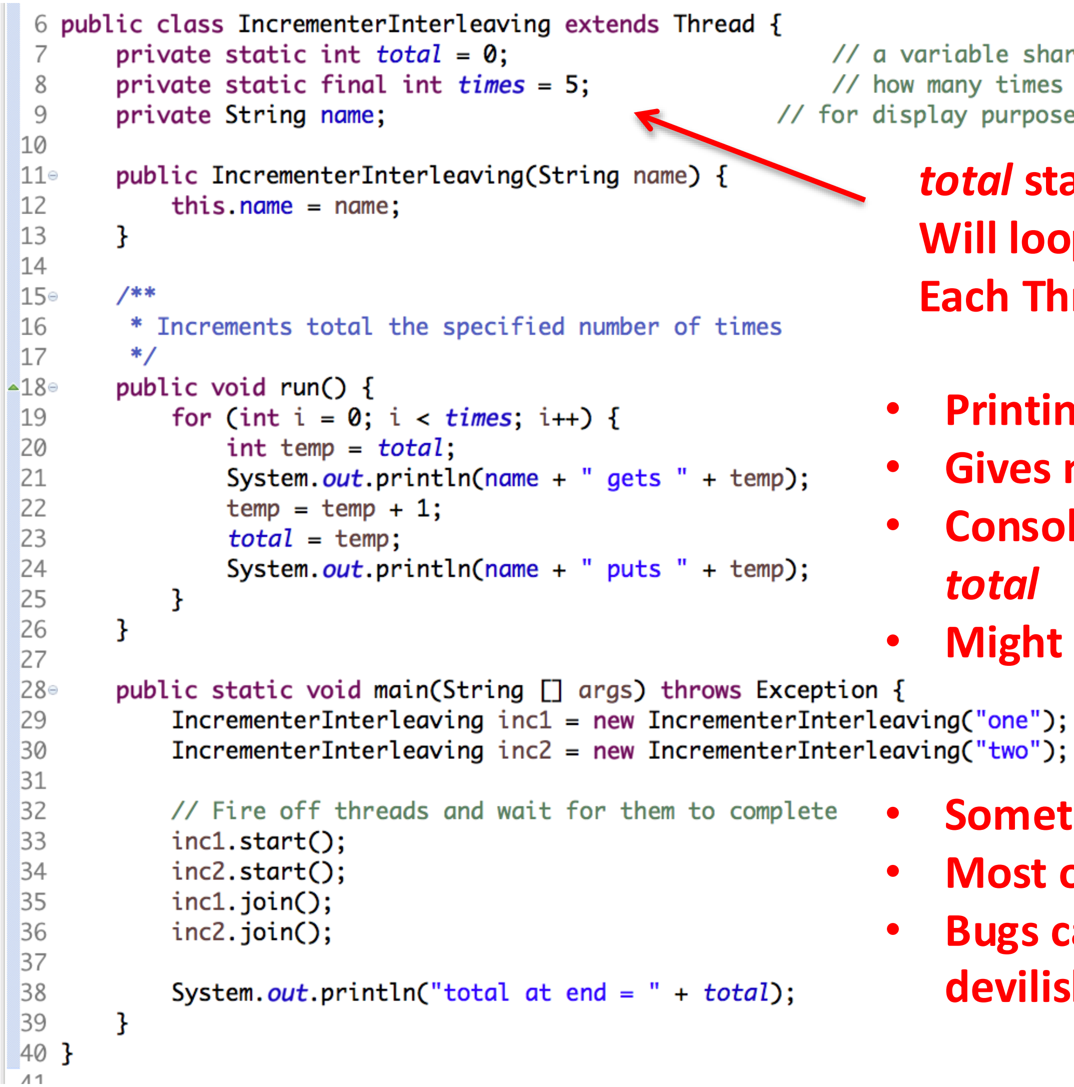
Operating System might interrupt a Thread at *any* point:

- ***inc1* reads value of *total* from memory (say it's 10)**
- ***inc1* gets interrupted and *inc2* begins running**
- ***inc2* reads value of *total* (10), increments and writes back (*total*=11)**

- **Say *inc2* runs for 5 iterations (*total*=15)**
- ***inc2* interrupted and *inc1* resumes running**
- ***inc1* increments *total* to 11 and writes it back**
- ***total* now 11 not 16 as expected**

IncrementerInterleaving.java demonstrates interruptions (sometimes)

IncrementerInterleaving.java



```java
public class IncrementerInterleaving extends Thread {
    private static int total = 0;                           // a variable shared by all incrementers
    private static final int times = 5;                     // how many times to increment total, in each thread
    private String name;                                // for display purposes

    public IncrementerInterleaving(String name) {
        this.name = name;
    }

    /**
     * Increments total the specified number of times
     */
    public void run() {
        for (int i = 0; i < times; i++) {
            int temp = total;
            System.out.println(name + " gets " + temp);
            temp = temp + 1;
            total = temp;
            System.out.println(name + " puts " + temp);
        }
    }

    public static void main(String [] args) throws Exception {
        IncrementerInterleaving inc1 = new IncrementerInterleaving("one");
        IncrementerInterleaving inc2 = new IncrementerInterleaving("two");

        // Fire off threads and wait for them to complete
        inc1.start();
        inc2.start();
        inc1.join();
        inc2.join();

        System.out.println("total at end = " + total);
    }
}
```

***total* static as before**

Will loop 5 times in *run()* method

Each Thread gets a name for clarity

- **Printing to console is *slooowwww***
- **Gives more time for OS to interrupt**
- **Console output shows when read and write *total***
- **Might expect total to be 10 (5 from inc1 and 5 from inc2)**

- **Sometimes total is 10**
- **Most of the time it is not**
- **Bugs caused by multiple threads can be devilishly tricky to find**

DEMO: IncrementerInterleaving.java

- Run several times
- Interrupted execution causes tricky bugs
- Sometimes it works as expected
- Most of the time it doesn’t…

Java provides the keyword synchronized to make some operations “atomic”

IncrementerTotal.java

```
public class IncrementerTotal {
    private int total = 0;
    public synchronized void inc() {
        total++;
    }
}
```

- **IncrementerTotal Class keeps a *total* instance variable**
- **Value of *total* incremented via *inc()* method**
- ***inc*() method is synchronized so only one Thread at a time can be inside *inc()***
- **IncrementerTotal Class used on next slide**

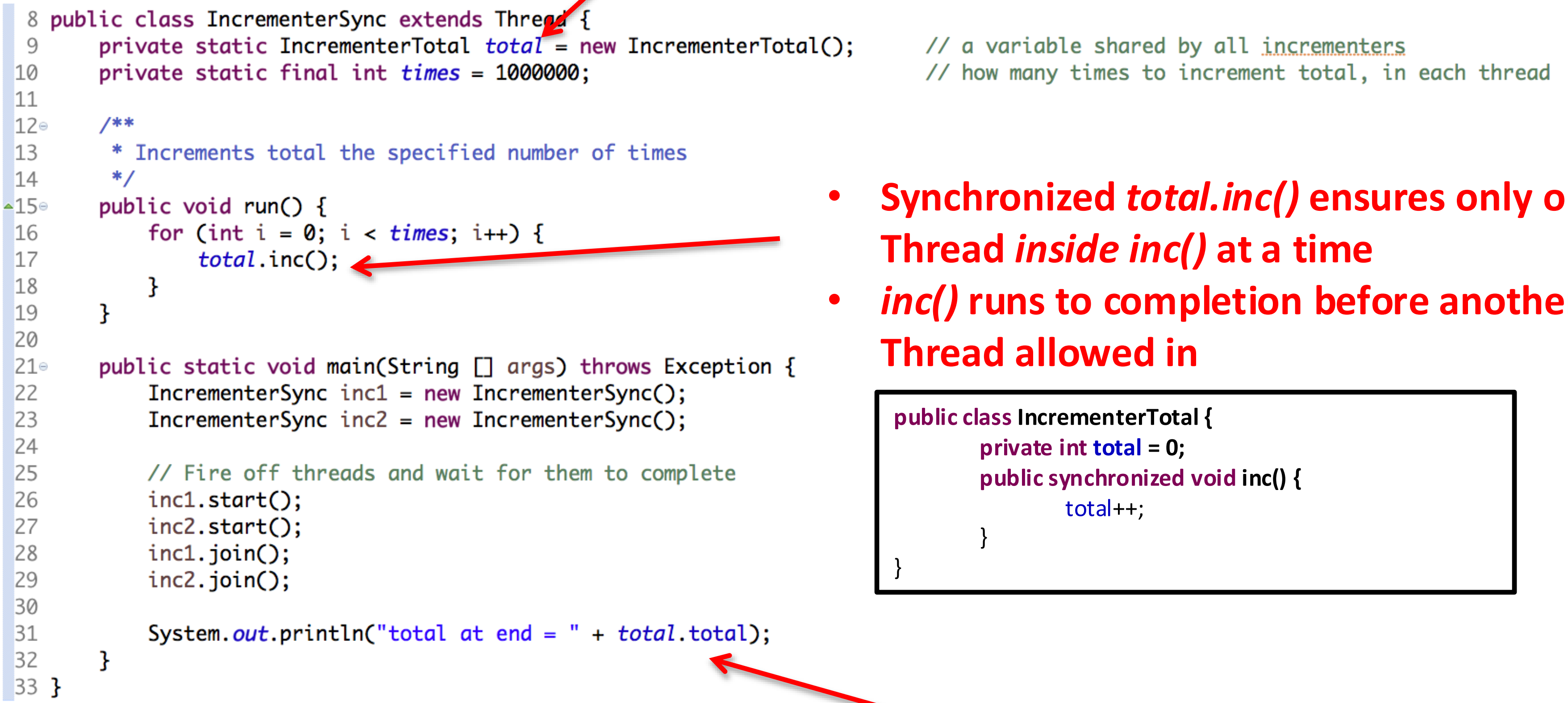
- `synchronized` keyword in front of `inc` method means only one thread can be running this code at a time
- If multiple threads try to run `synchronized` code, one thread runs, all others block until first one finishes
- Once first thread finishes, OS selects another thread to run
- `synchronized` makes this code “atomic” (e.g., as if it were one instruction)
- This synchronized approach is called a “mutex” (or monitor), acts like a “lock” on static `total` variable

IncrementerSync.java uses atomic operations to ensure desired behavior

IncrementerSync.java

***total* now an IncrementerTotal Object**

***total.inc()* is synchronized**



```
public class IncrementerSync extends Thread {
    private static IncrementerTotal total = new IncrementerTotal();      // a variable shared by all incrementers
    private static final int times = 1000000;                           // how many times to increment total, in each thread

    /**
     * Increments total the specified number of times
     */
    public void run() {
        for (int i = 0; i < times; i++) {
            total.inc();
        }
    }

    public static void main(String [] args) throws Exception {
        IncrementerSync inc1 = new IncrementerSync();
        IncrementerSync inc2 = new IncrementerSync();

        // Fire off threads and wait for them to complete
        inc1.start();
        inc2.start();
        inc1.join();
        inc2.join();

        System.out.println("total at end = " + total.total);
    }
}
```

- **Synchronized *total.inc()* ensures only one Thread *inside inc()* at a time**
- ***inc()* runs to completion before another Thread allowed in**

```
public class IncrementerTotal {
        private int total = 0;
        public synchronized void inc() {
                total++;
        }
}
```

***total.total* now always 2 million**

Agenda

1. Interleaving execution
2. Producer/consumer
3. Deadlock, starvation

Producers tell Consumers when ready, Consumers tell Producers when done

Big idea: keep Producers and Consumers in sync

Producer:

- Tell Consumer when item is ready (`notify` or `notifyAll`)
- Block until woken up by Consumer that item handled (`wait`)
- Tell Consumer when next item is ready (`notify` or `notifyAll`)
- There can be multiple Producers

Consumer:

- Block until woken up by Producer that item ready (`wait`)
- Process item and tell Producer when done (`notify` or `notifyAll`)
- Block until woken up by Producer (`wait`)
- There can be multiple Consumers

Producers and Consumers synchronized with `wait`, `notify` or `notifyAll`

`wait()`

- Pauses and removes Thread from synchronized method
- Tells Operating System to put this Thread into a list of Threads waiting to resume execution
- `wait()` allows another Thread to enter synchronized method

`notify()`

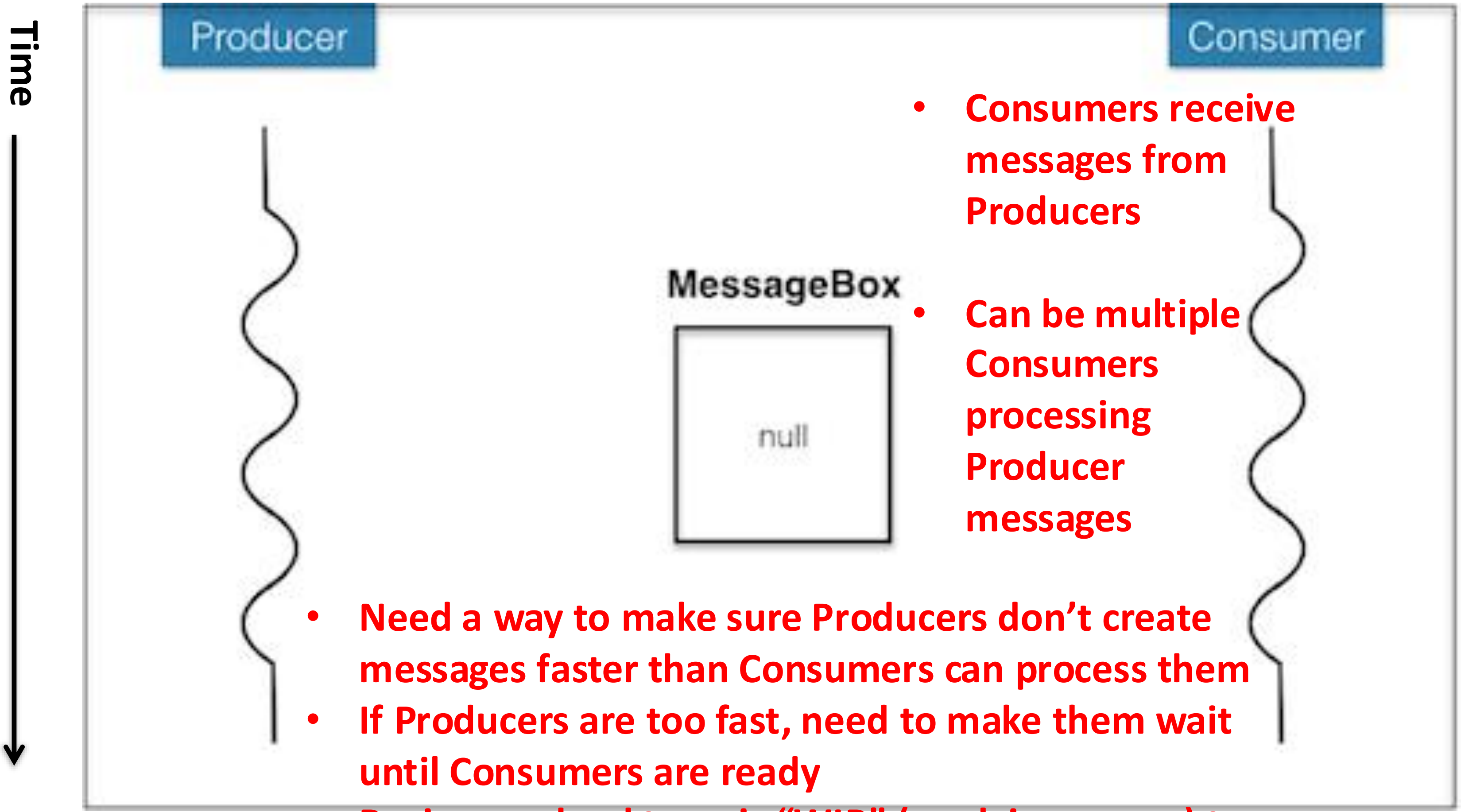
- Tells Operating System to pick a waiting Thread and let it run again (not a FIFO queue, OS decides – take CS58 for more)
- Thread should check that conditions are met for it to continue

`notifyAll()`

- Wake up all waiting Threads
- Each Thread should check that conditions are met for it to continue

Scenario: Producers produce messages for Consumers, need to keep in sync

Example

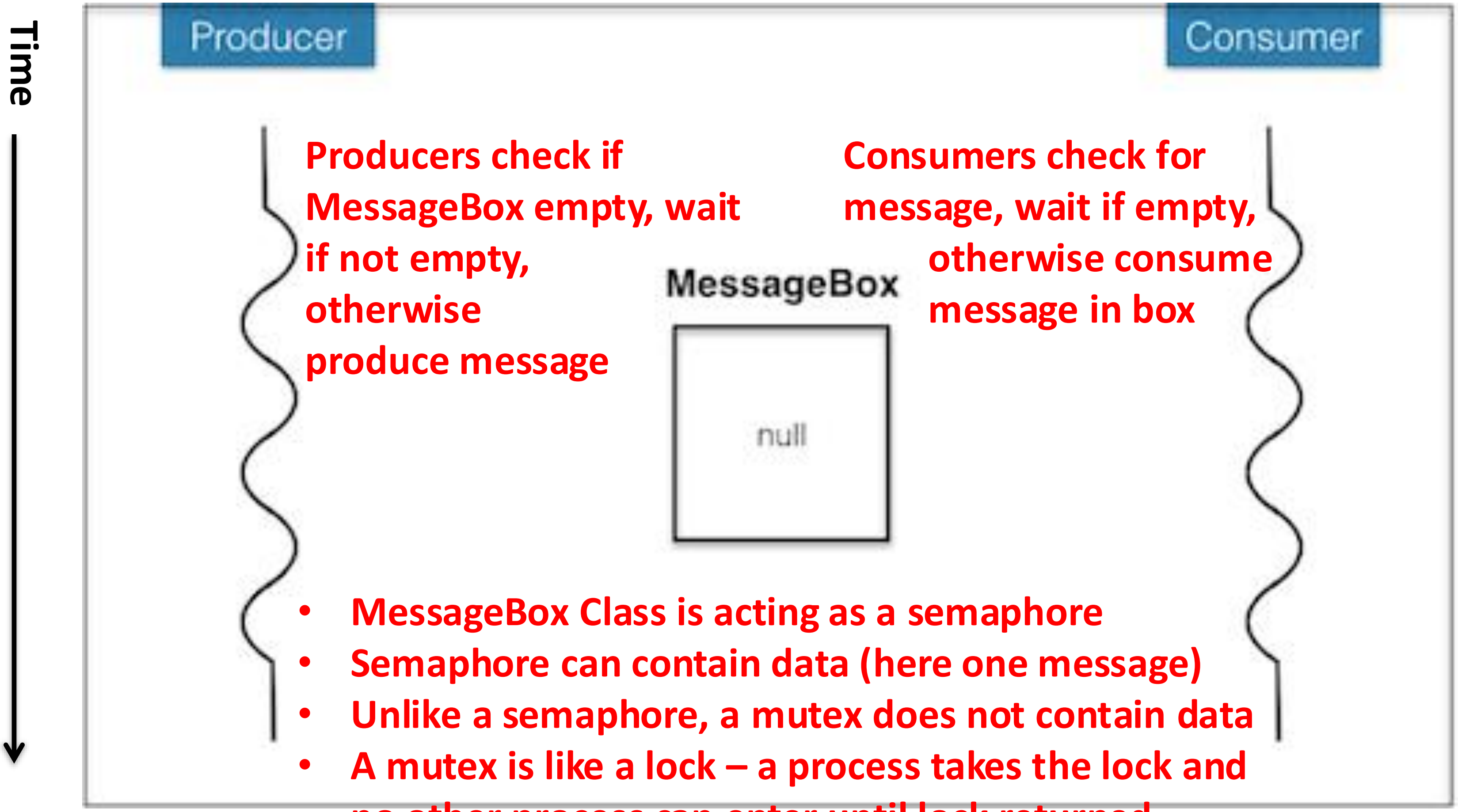


- Consumers receive messages from Producers
- Can be multiple Consumers processing Producer messages

- Need a way to make sure Producers don't create messages faster than Consumers can process them
- If Producers are too fast, need to make them wait until Consumers are ready
- Business school term is "WIP" (work in process) to describe items built up if Producers generate items faster than Consumers handle them

We can use a semaphore to keep Producers and Consumers in sync

Example



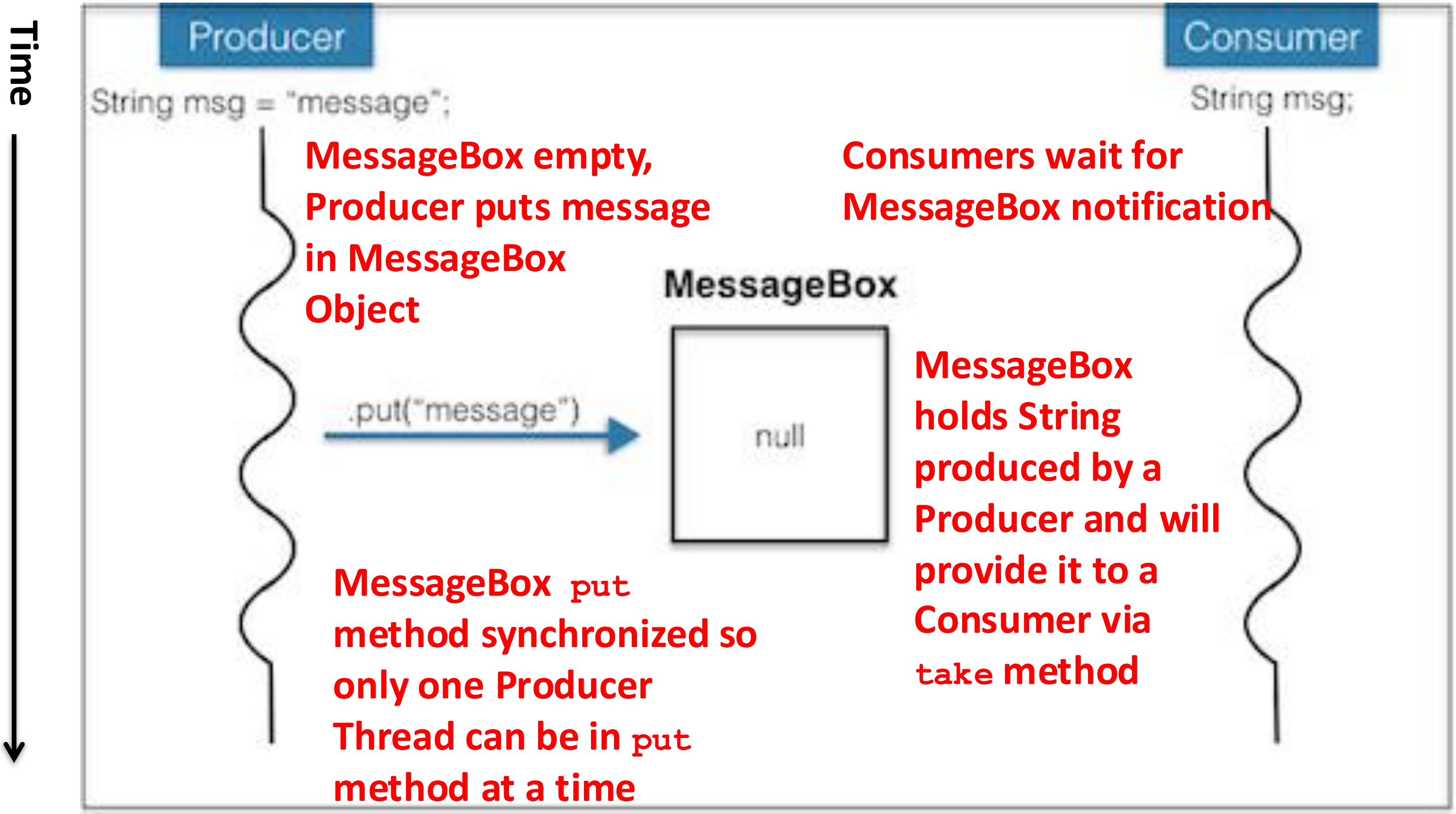
Producers check if MessageBox empty, wait if not empty, otherwise produce message

Consumers check for message, wait if empty, otherwise consume message in box

- MessageBox Class is acting as a semaphore
- Semaphore can contain data (here one message)
- Unlike a semaphore, a mutex does not contain data
- A mutex is like a lock – a process takes the lock and no other process can enter until lock returned

Producer passing messages to Consumer using semaphore

Example



MessageBox empty, Producer puts message in MessageBox Object

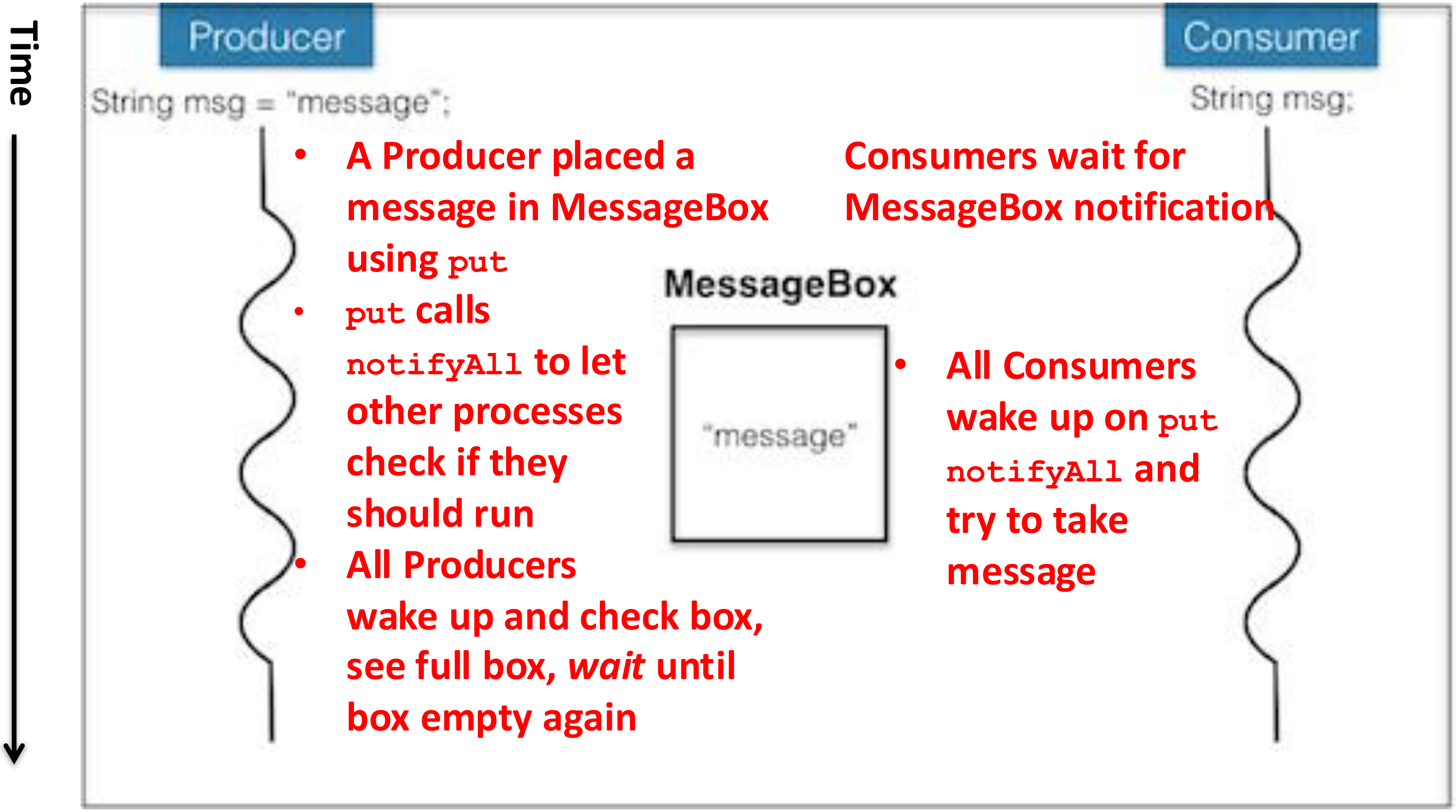
Consumers wait for MessageBox notification

MessageBox holds String produced by a Producer and will provide it to a Consumer via `take` method

MessageBox `put` method synchronized so only one Producer Thread can be in `put` method at a time

Producer passing messages to Consumer using semaphore

Example



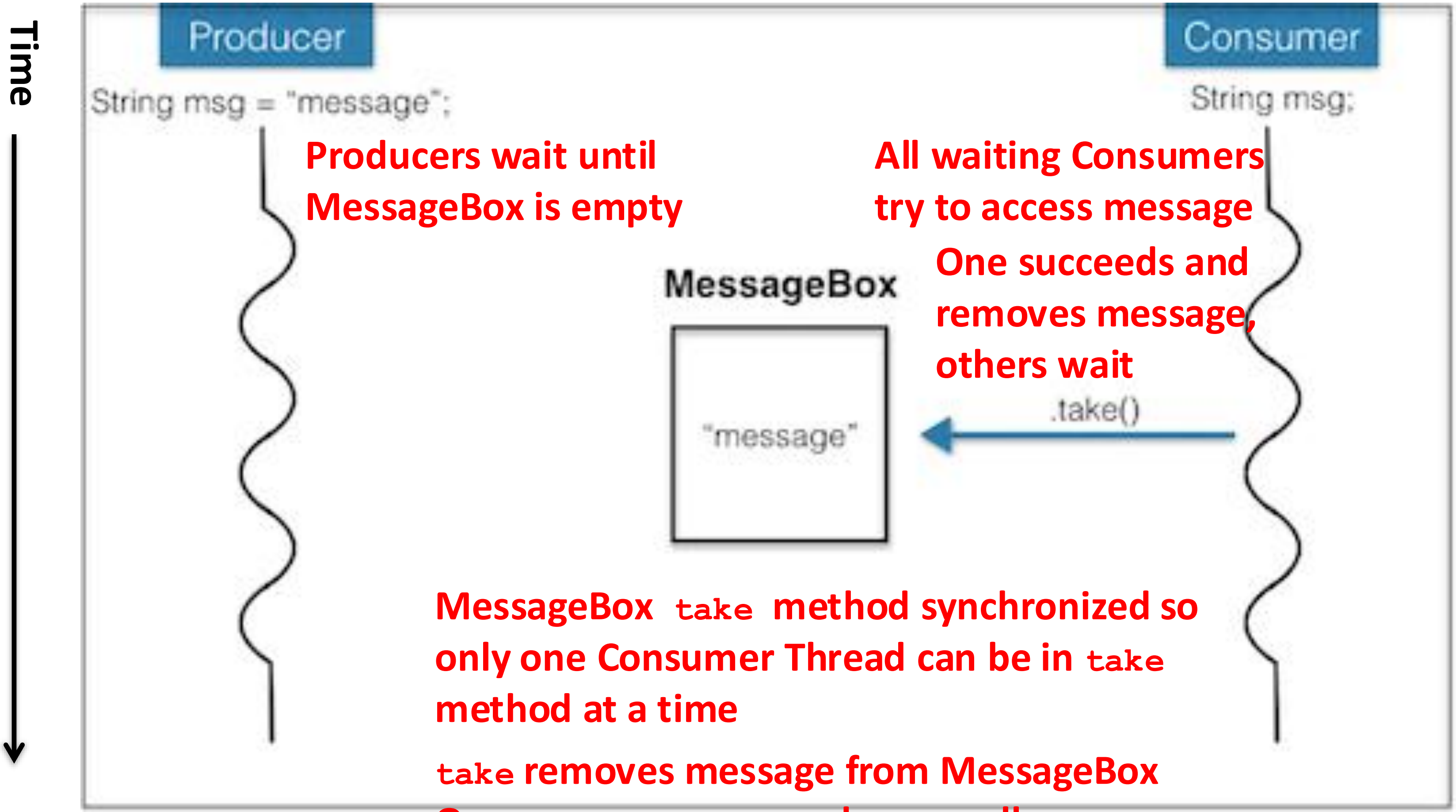
- A Producer placed a message in MessageBox using `put`
- `put` calls `notifyAll` to let other processes check if they should run
- All Producers wake up and check box, see full box, *wait* until box empty again

Consumers wait for MessageBox notification

- All Consumers wake up on `put` `notifyAll` and try to take message

Producer passing messages to Consumer using semaphore

Example



Producers wait until MessageBox is empty

All waiting Consumers try to access message

One succeeds and removes message, others wait

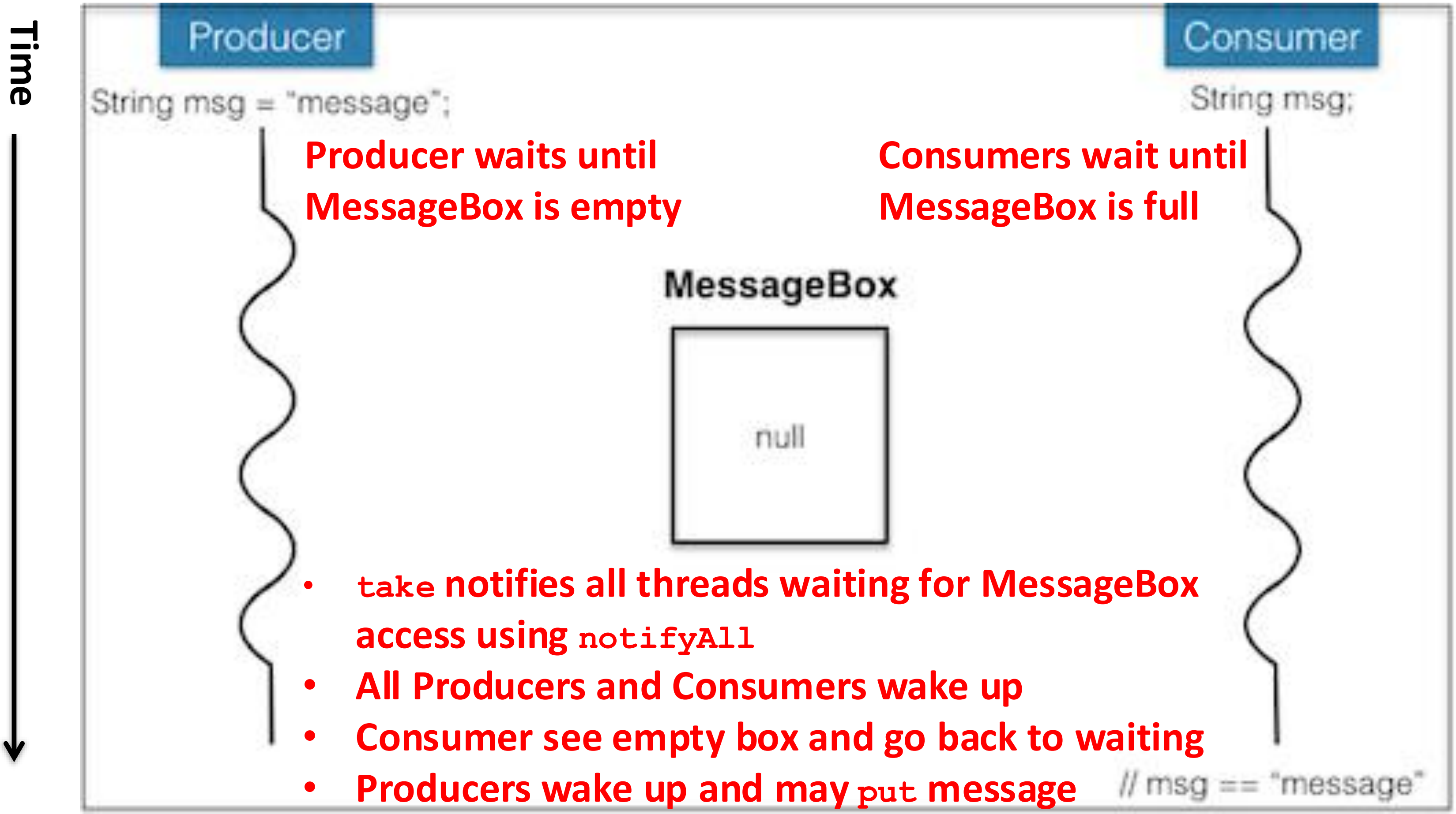
MessageBox `take` method synchronized so only one Consumer Thread can be in `take` method at a time

`take` removes message from MessageBox

Once message removed, `take` calls `notifyAll` to let other processes check if they should run

Producer passing messages to Consumer using semaphore

Example



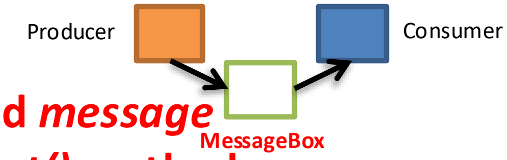
Producer waits until MessageBox is empty

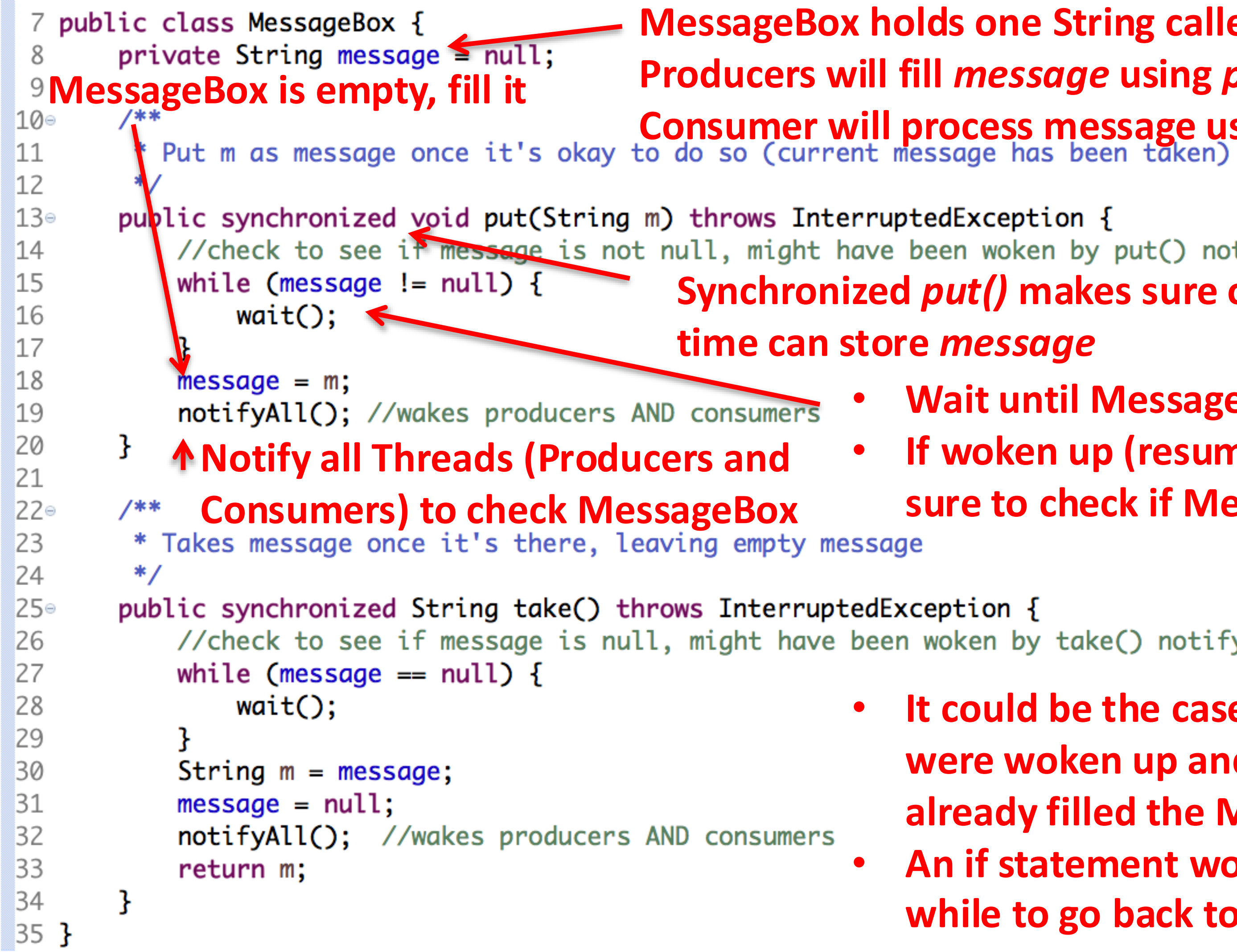
Consumers wait until MessageBox is full

- `take` notifies all threads waiting for MessageBox access using `notifyAll`
- All Producers and Consumers wake up
- Consumer see empty box and go back to waiting
- Producers wake up and may `put` message now, one succeeds and others go back to waiting
- Process repeats with Producers and Consumer in sync

MessageBox.java implements a semaphore that holds one String

MessageBox.java





```java
public class MessageBox {
    private String message = null;

    /**
     * Put m as message once it's okay to do so (current message has been taken)
     */
    public synchronized void put(String m) throws InterruptedException {
        //check to see if message is not null, might have been woken by put() notifyAll
        while (message != null) {
            wait();
        }
        message = m;
        notifyAll(); //wakes producers AND consumers
    }

    /**
     * Takes message once it's there, leaving empty message
     */
    public synchronized String take() throws InterruptedException {
        //check to see if message is null, might have been woken by take() notifyAll
        while (message == null) {
            wait();
        }
        String m = message;
        message = null;
        notifyAll();  //wakes producers AND consumers
        return m;
    }
}
```

MessageBox holds one String called *message*

Producers will fill *message* using *put()* method

Consumer will process message using *take()* method

MessageBox is empty, fill it

Synchronized *put()* makes sure only one Producer at a time can store *message*

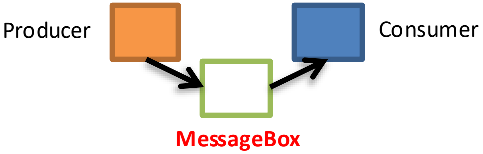
- **Wait until MessageBox is empty**
- **If woken up (resume running at *wait*), make sure to check if MessageBox is empty**

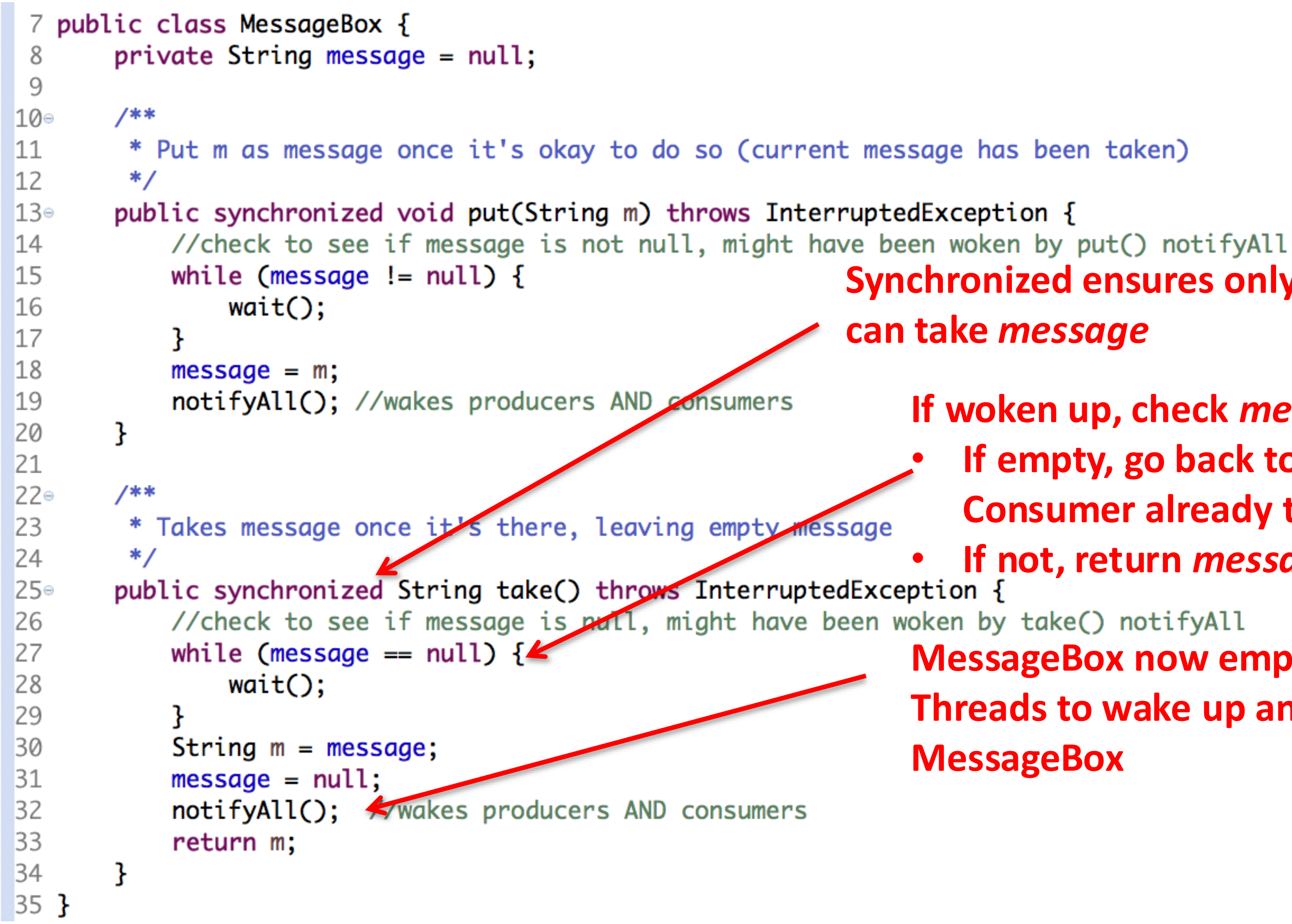
Notify all Threads (Producers and Consumers) to check MessageBox

- **It could be the case that many Producers were woken up and another Producer already filled the MessageBox**
- **An if statement wouldn't suffice, need a while to go back to sleep if box filled**

MessageBox.java implements a semaphore that holds one String

MessageBox.java





```java
public class MessageBox {
    private String message = null;

    /**
     * Put m as message once it's okay to do so (current message has been taken)
     */
    public synchronized void put(String m) throws InterruptedException {
        //check to see if message is not null, might have been woken by put() notifyAll
        while (message != null) {
            wait();
        }
        message = m;
        notifyAll(); //wakes producers AND consumers
    }

    /**
     * Takes message once it's there, leaving empty message
     */
    public synchronized String take() throws InterruptedException {
        //check to see if message is null, might have been woken by take() notifyAll
        while (message == null) {
            wait();
        }
        String m = message;
        message = null;
        notifyAll(); //wakes producers AND consumers
        return m;
    }
}
```

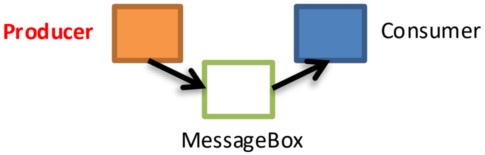
Synchronized ensures only one Consumer can take *message*

If woken up, check *message*:

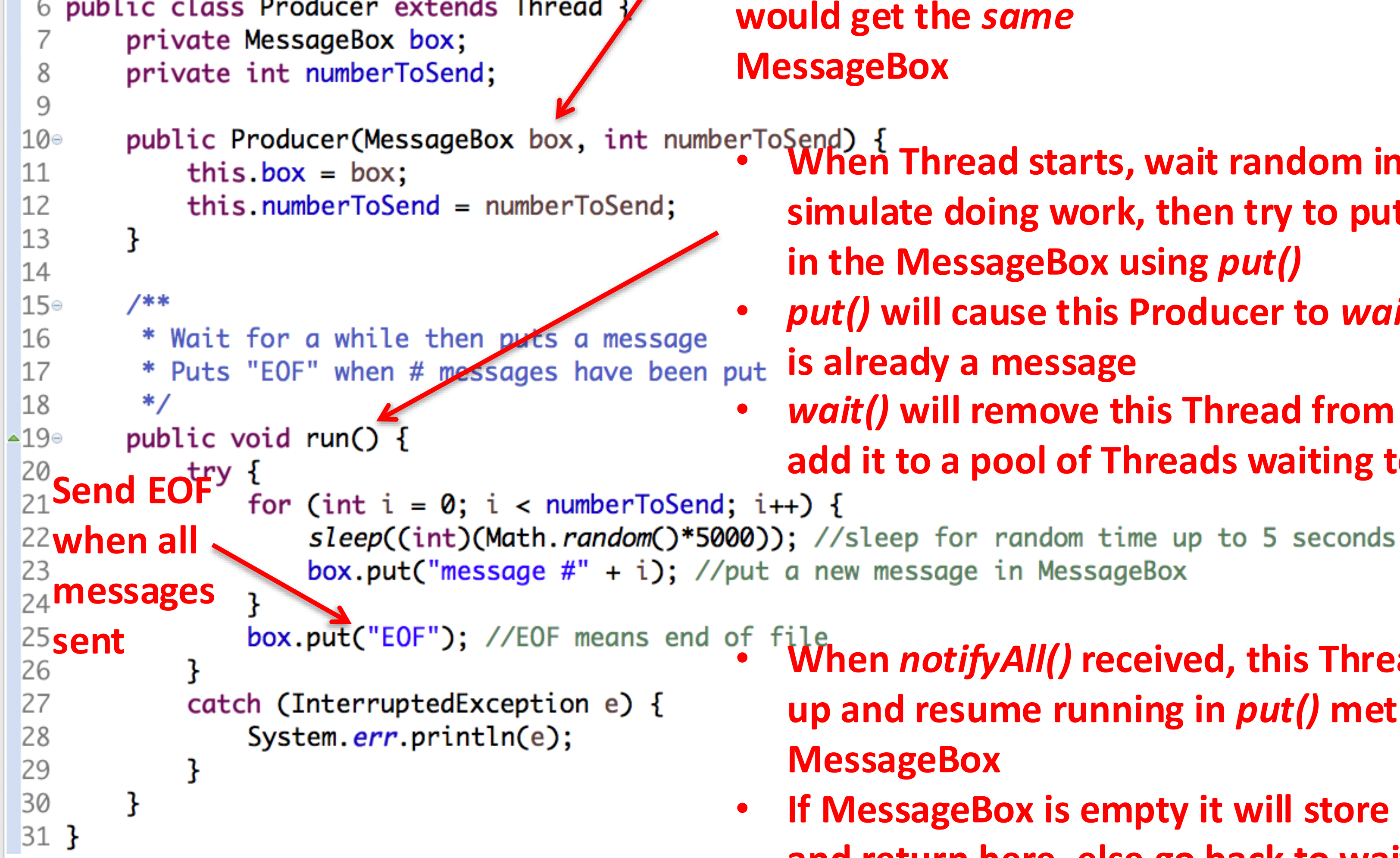
- **If empty, go back to waiting (another Consumer already took it)**
- **If not, return *message* and set to null**

MessageBox now empty, notify all Threads to wake up and check MessageBox

Producers use MessageBox to pass messages to Consumers



Producer.java



```java
public class Producer extends Thread {
    private MessageBox box;
    private int numberToSend;

    public Producer(MessageBox box, int numberToSend) {
        this.box = box;
        this.numberToSend = numberToSend;
    }

    /**
     * Wait for a while then puts a message
     * Puts "EOF" when # messages have been put
     */
    public void run() {
        try {
            for (int i = 0; i < numberToSend; i++) {
                sleep((int)(Math.random()*5000)); //sleep for random time up to 5 seconds
                box.put("message #" + i); //put a new message in MessageBox
            }
            box.put("EOF"); //EOF means end of file
        }
        catch (InterruptedException e) {
            System.err.println(e);
        }
    }
}
```

MessageBox as parameter

If multiple Producers, all would get the *same* MessageBox

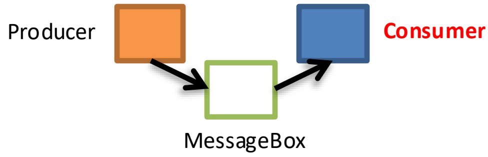
- When Thread starts, wait random interval to simulate doing work, then try to put a *message* in the MessageBox using *put()*
- *put()* will cause this Producer to *wait()* if there is already a message
- *wait()* will remove this Thread from *put()* and add it to a pool of Threads waiting to run

Send EOF when all messages sent

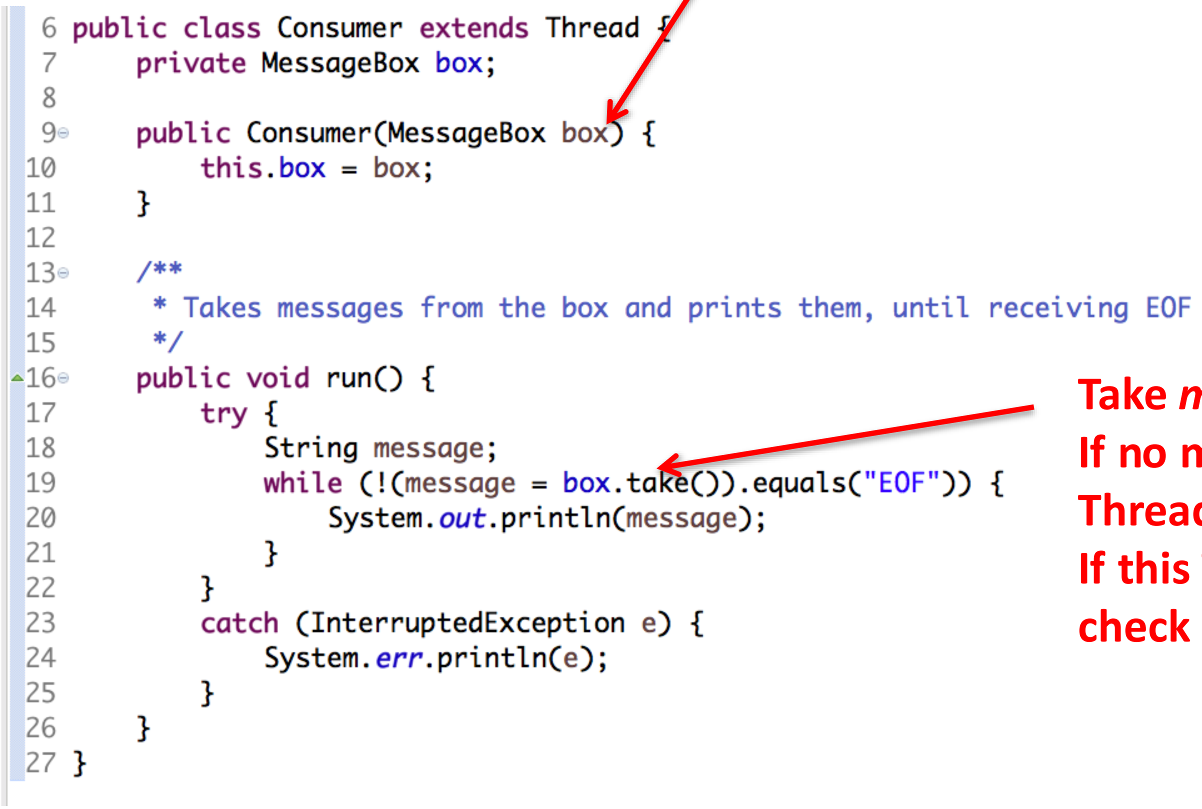
- When *notifyAll()* received, this Thread will wake up and resume running in *put()* method of MessageBox
- If MessageBox is empty it will store it's message and return here, else go back to waiting

Consumers retrieve messages from the MessageBox

Consumer.java



Store same MessageBox that Producers use

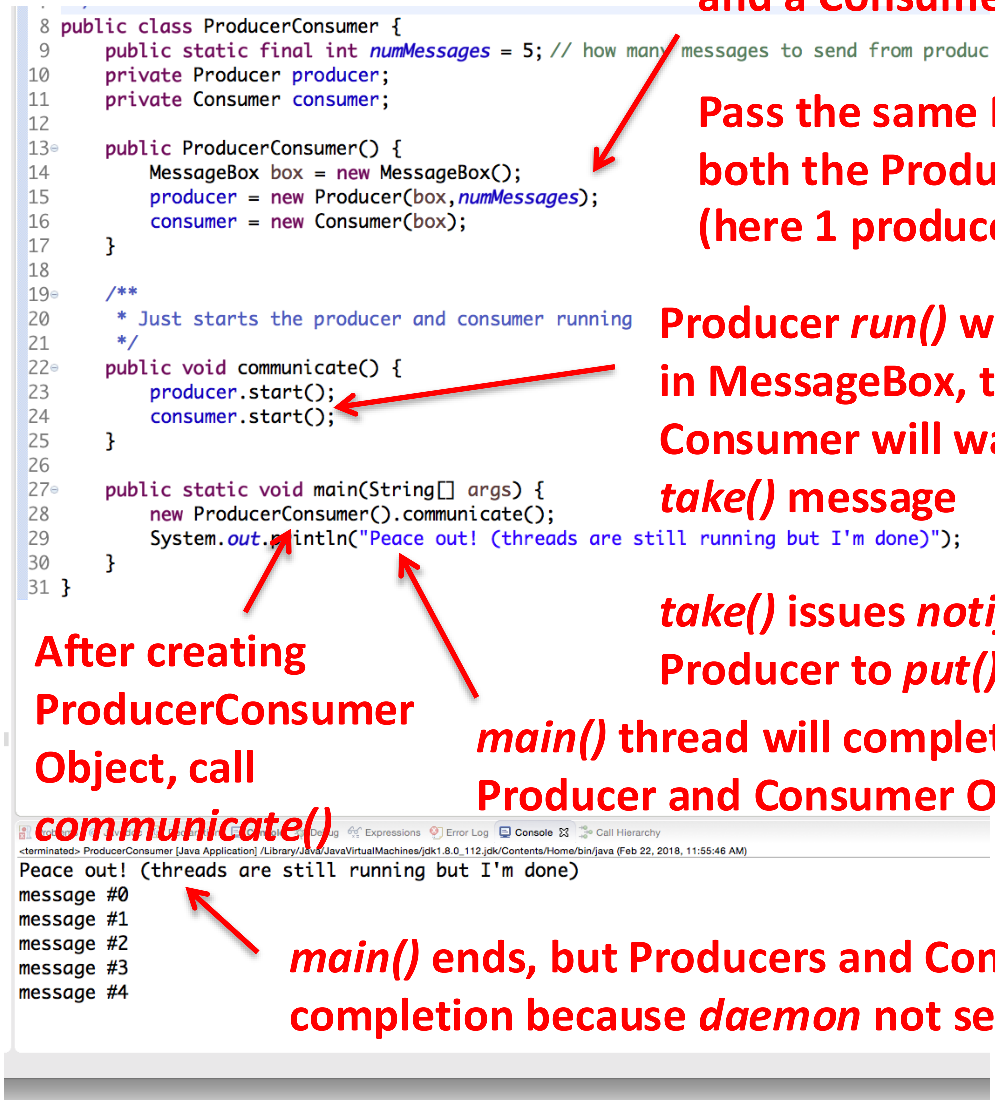


```
public class Consumer extends Thread {
    private MessageBox box;

    public Consumer(MessageBox box) {
        this.box = box;
    }

    /**
     * Takes messages from the box and prints them, until receiving EOF
     */
    public void run() {
        try {
            String message;
            while (!(message = box.take()).equals("EOF")) {
                System.out.println(message);
            }
        }
        catch (InterruptedException e) {
            System.err.println(e);
        }
    }
}
```

Take *message* from MessageBox
If no message, *take()* will cause this Thread to wait
If this Thread retrieves message, check for EOF and exit

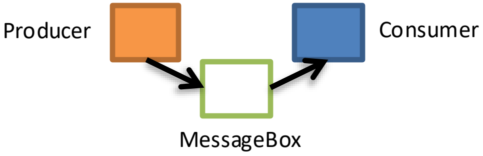
ProducerConsumer uses all three components to pass messages

ProducerConsumer.java



```
public class ProducerConsumer {
    public static final int numMessages = 5; // how many messages to send from produc
    private Producer producer;
    private Consumer consumer;

    public ProducerConsumer() {
        MessageBox box = new MessageBox();
        producer = new Producer(box,numMessages);
        consumer = new Consumer(box);
    }

    /**
     * Just starts the producer and consumer running
     */
    public void communicate() {
        producer.start();
        consumer.start();
    }

    public static void main(String[] args) {
        new ProducerConsumer().communicate();
        System.out.println("Peace out! (threads are still running but I'm done)");
    }
}
```

```
<terminated> ProducerConsumer [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_112.jdk/Contents/Home/bin/java (Feb 22, 2018, 11:55:46 AM)
Peace out! (threads are still running but I'm done)
message #0
message #1
message #2
message #3
message #4
```



Create a MessageBox, a Producer, and a Consumer

Pass the same MessageBox Object to both the Producer and the Consumer (here 1 producer and 1 consumer)

Producer *run()* will wait a random period, then put a *message* in MessageBox, then wait until MessageBox empty

Consumer will wake up on *notifyAll()* from MessageBox and *take()* message

take()* issues *notifyAll()* after taking *message*, waking Producer to *put()* next *message

After creating ProducerConsumer Object, call *communicate()*

***main()* thread will complete after starting both Producer and Consumer Objects**

***main()* ends, but Producers and Consumers threads run to completion because *daemon* not set to true**

Agenda

1. Interleaving execution
2. Producer/consumer
3. Deadlock, starvation

Synchronization can lead to two problems: deadlock and starvation

Deadlock

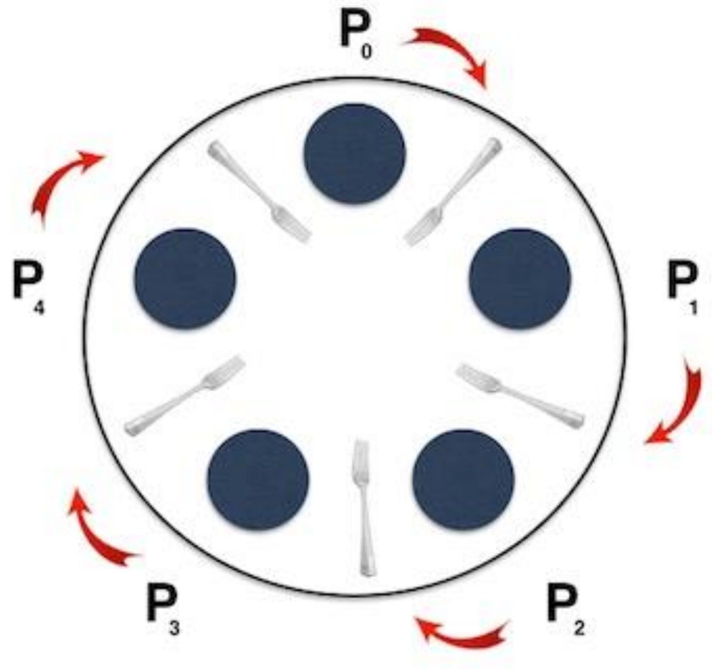
- Objects lock resources
- Execution cannot proceed because object needs a resource another locked
- Object A locks resource 1
- Object B locks resource 2
- A needs resource 2 to proceed but B has it locked
- B needs resources 1 to proceed but A has it locked
- A and B are deadlocked

Starvation

- Thread never gets resource it needs
- Thread A needs resource 1 to complete
- Other threads always take resource 1 before A can get it
- We say A is *starved*

Dinning Philosophers explains deadlock and starvation

Dinning Philosophers

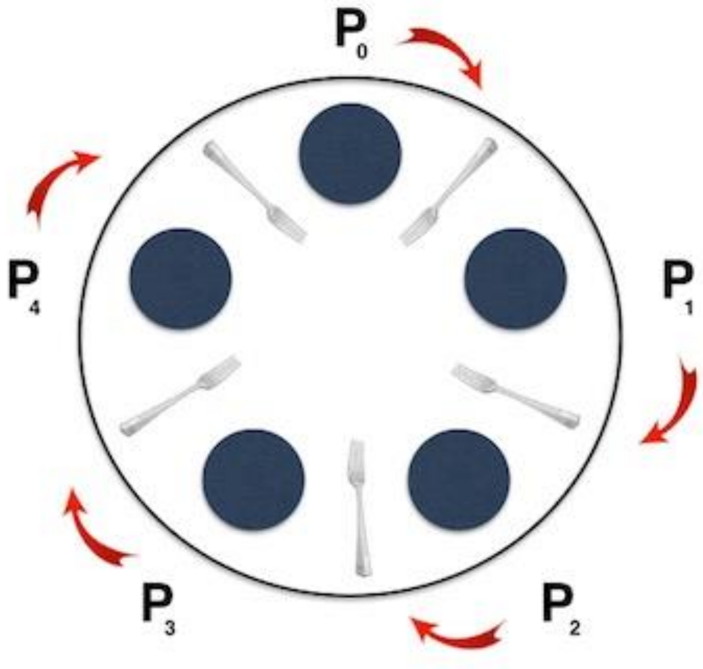


Problem set up

- Five philosophers (P_0-P_4) sit at a table to eat spaghetti
- There are forks between each of them (five total forks)
- Each philosopher needs two forks to eat
- After acquiring two forks, philosopher eats, then puts both forks down
- Another philosopher can then pick up and use fork previously put down (gross!)

Dinning Philosophers explains deadlock and starvation

Dinning Philosophers



Naïve approach

- Each philosopher picks up fork on left
- Then picks up fork on right
- Deadlock occurs if all philosophers get left fork, none get right fork

For deadlock to occur four conditions must be met

Deadlock conditions

1. **Mutual exclusion**
    - At least one resource class must have non-sharable access. That is:
        - Either one process is using a resource (and others wait), or
        - Resource is free
2. **Hold and wait**
    - At least one process is holding a resource instance, while also waiting to be granted another resource instance. (e.g., Each philosopher is holding on to their left fork, while waiting to pick up their right fork)
3. **No preemption**
    - Resources cannot be pre-empted; a resource can be released only voluntarily by the process holding it (e.g., can't force philosophers to drop their forks.)
4. **Circular wait**
    - There must exist a circular chain of at least two processes, each of whom is waiting for a resource held by another one. (e.g., each Philosopher[i] is waiting for Philosopher[(i+1) mod 5] to drop its fork.)

From Coffman, 1971

Three ways to ensure deadlock does not occur

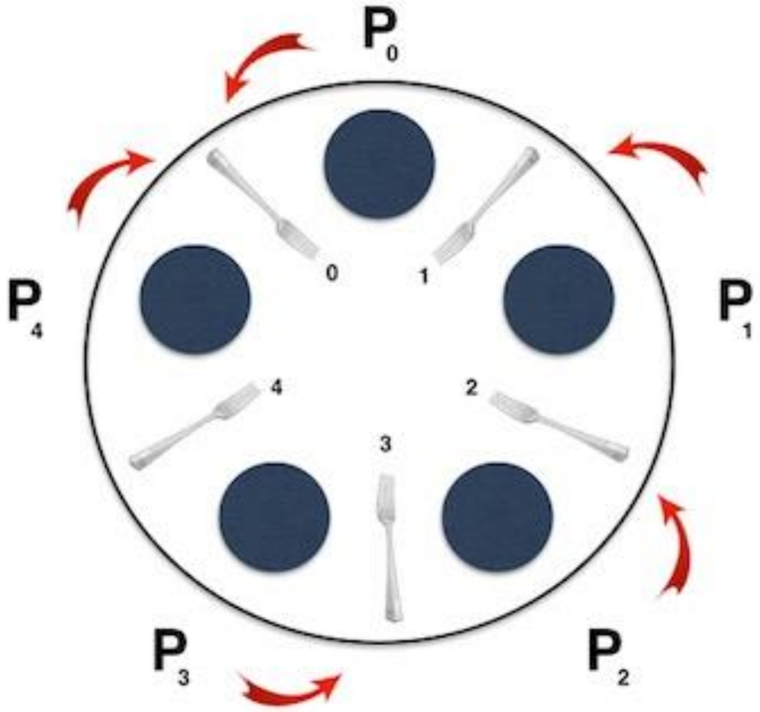
1. Ensure circular wait cannot occur by numbering Forks and reaching for smallest numbered Fork first

2. Prevent circular wait by making one of the philosophers wait until at least one other philosopher is finished

3. Prevent hold and wait by making Fork acquisition an atomic operation (e.g., must get both Forks in one step)

We can break the deadlock by ensuring the "circular wait" does not occur

Dinning Philosophers



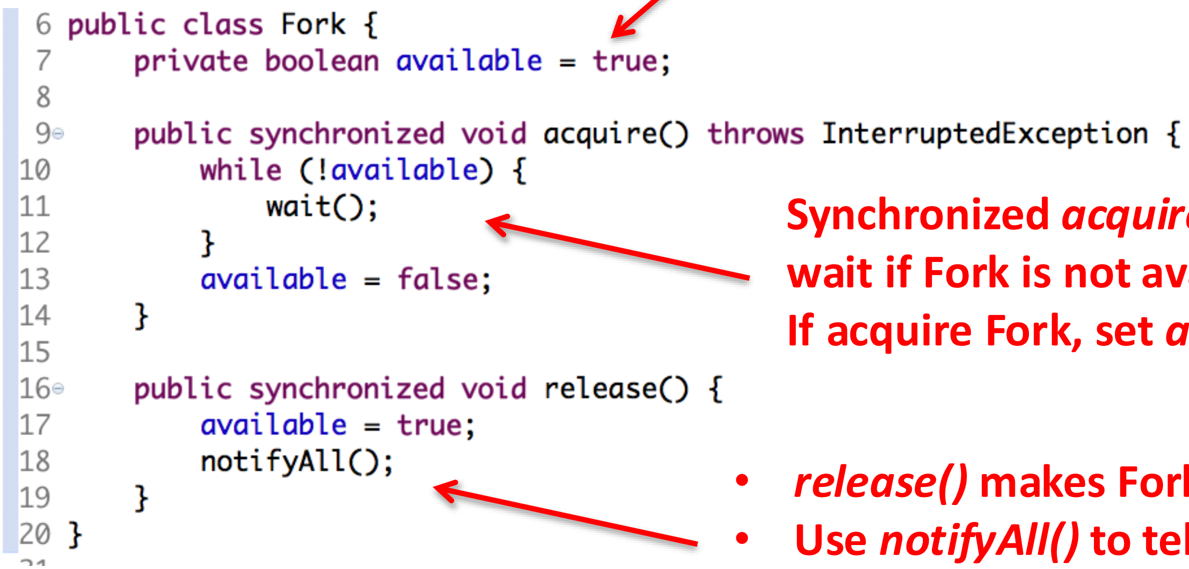
Could also force one of the Philosophers to wait at first

Eliminate circular wait

- Number each fork in circular fashion
- Make each philosopher pick up lowest numbered fork first
- All pick up right fork, except P_4 who tries to pick up left fork 0
- Either P_0 or P_4 get fork 0
- If P_0 gets it, P_4 waits for fork 0 before picking up fork 4, so P_3 eats
- P_3 eventually releases both forks and P_2 eats
- Others eat after P_2
- Cannot deadlock

Fork.java models forks in the Dining Philosophers problem

Fork.java



```java
public class Fork {
    private boolean available = true;

    public synchronized void acquire() throws InterruptedException {
        while (!available) {
            wait();
        }
        available = false;
    }

    public synchronized void release() {
        available = true;
        notifyAll();
    }
}
```

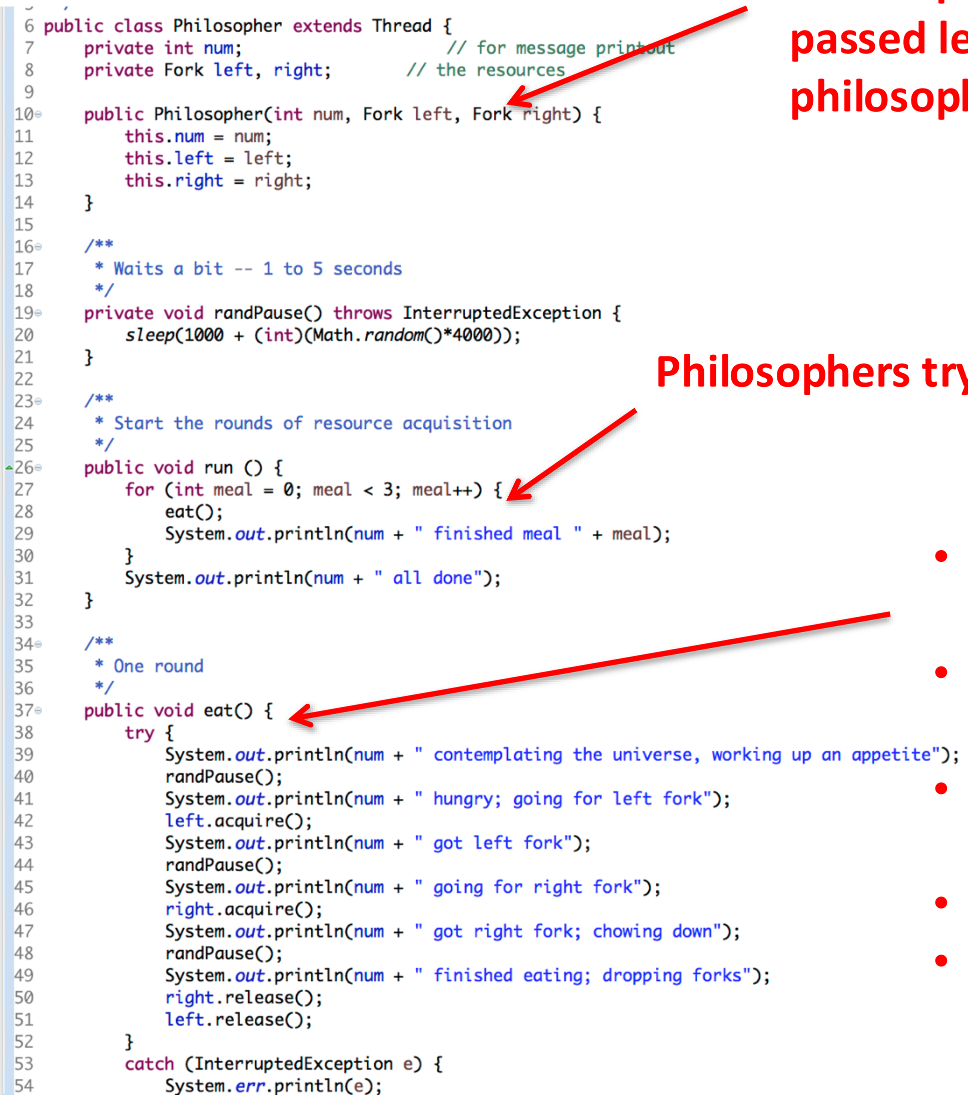
available **tracks if this Fork Object is being used**

Synchronized *acquire()* causes wait if Fork is not available
If acquire Fork, set *available* false

- ***release()*** **makes Fork available to others**
- **Use *notifyAll()* to tell Philosophers a Fork is free**

Philosophers try to eat by getting both the left and right Forks

Philosopher.java



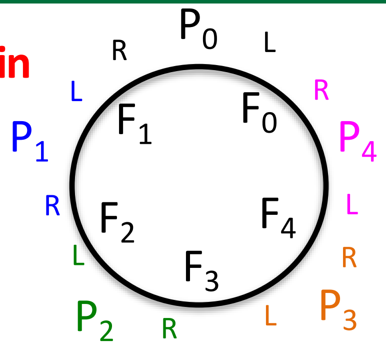
```java
public class Philosopher extends Thread {
    private int num;                    // for message printout
    private Fork left, right;       // the resources

    public Philosopher(int num, Fork left, Fork right) {
        this.num = num;
        this.left = left;
        this.right = right;
    }

    /**
     * Waits a bit -- 1 to 5 seconds
     */
    private void randPause() throws InterruptedException {
        sleep(1000 + (int)(Math.random()*4000));
    }

    /**
     * Start the rounds of resource acquisition
     */
    public void run () {
        for (int meal = 0; meal < 3; meal++) {
            eat();
            System.out.println(num + " finished meal " + meal);
        }
        System.out.println(num + " all done");
    }

    /**
     * One round
     */
    public void eat() {
        try {
            System.out.println(num + " contemplating the universe, working up an appetite");
            randPause();
            System.out.println(num + " hungry; going for left fork");
            left.acquire();
            System.out.println(num + " got left fork");
            randPause();
            System.out.println(num + " going for right fork");
            right.acquire();
            System.out.println(num + " got right fork; chowing down");
            randPause();
            System.out.println(num + " finished eating; dropping forks");
            right.release();
            left.release();
        }
        catch (InterruptedException e) {
            System.err.println(e);
```

Philosopher runs on a Thread and is passed left and right Fork (also passed a philosopher number)

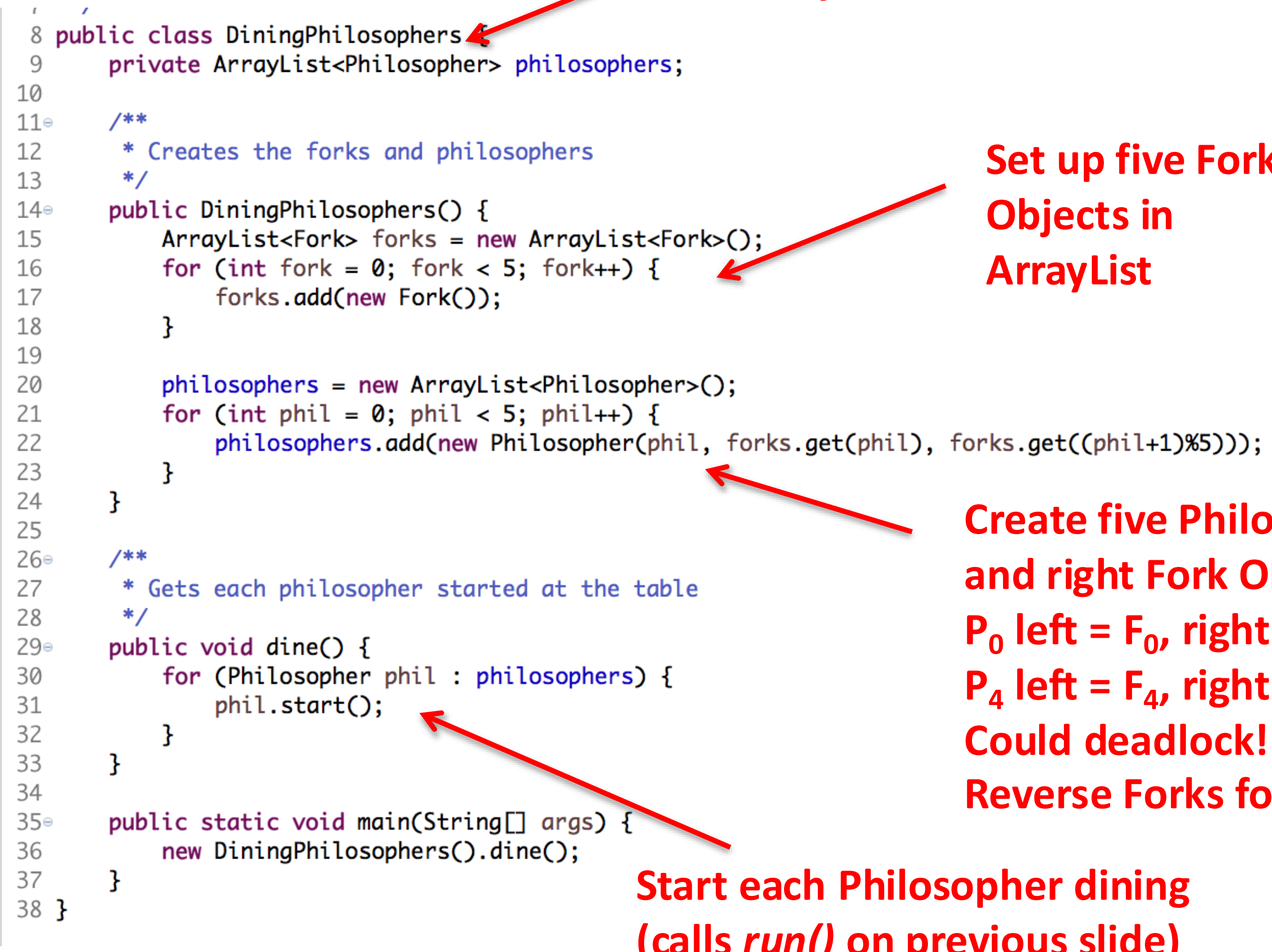
Philosophers try to eat three meals

- ***eat()* tries to *acquire()* the left and right fork (after universe contemplation of course)**
- **Always tries to get Fork on left first (could be a problem if Forks not numbered properly)**
- ***acquire()* will cause a wait if Fork not available**
- **Once philosopher has both Forks, he can eat**
- **Philosopher releases both Forks after eating**

DiningPhilosophers.java uses five Philosophers and five Forks



DiningPhilosopher.java



```java
public class DiningPhilosophers {
    private ArrayList<Philosopher> philosophers;

    /**
     * Creates the forks and philosophers
     */
    public DiningPhilosophers() {
        ArrayList<Fork> forks = new ArrayList<Fork>();
        for (int fork = 0; fork < 5; fork++) {
            forks.add(new Fork());
        }

        philosophers = new ArrayList<Philosopher>();
        for (int phil = 0; phil < 5; phil++) {
            philosophers.add(new Philosopher(phil, forks.get(phil), forks.get((phil+1)%5)));
        }
    }

    /**
     * Gets each philosopher started at the table
     */
    public void dine() {
        for (Philosopher phil : philosophers) {
            phil.start();
        }
    }

    public static void main(String[] args) {
        new DiningPhilosophers().dine();
    }
}
```

Will hold multiple Philosophers in ArrayList

Set up five Fork Objects in ArrayList

Create five Philosophers and pass the left and right Fork Objects

P_0 left = F_0, right = F_1

P_4 left = F_4, right = F_0

Could deadlock!

Reverse Forks for P_4 and won’t deadlock

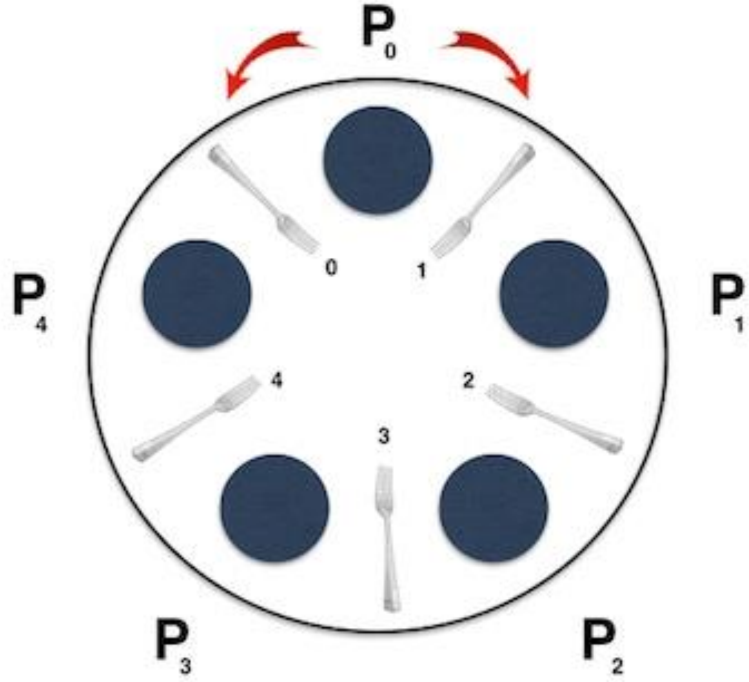
Start each Philosopher dining (calls *run()* on previous slide)

DEMO: DiningPhilosophers.java

- Run several times
- Sometimes deadlocks
- Try adjusting pause time to longer to make it less likely to deadlock

Another approach is to prevent “hold and wait” by picking up both forks atomically

Dinning Philosophers

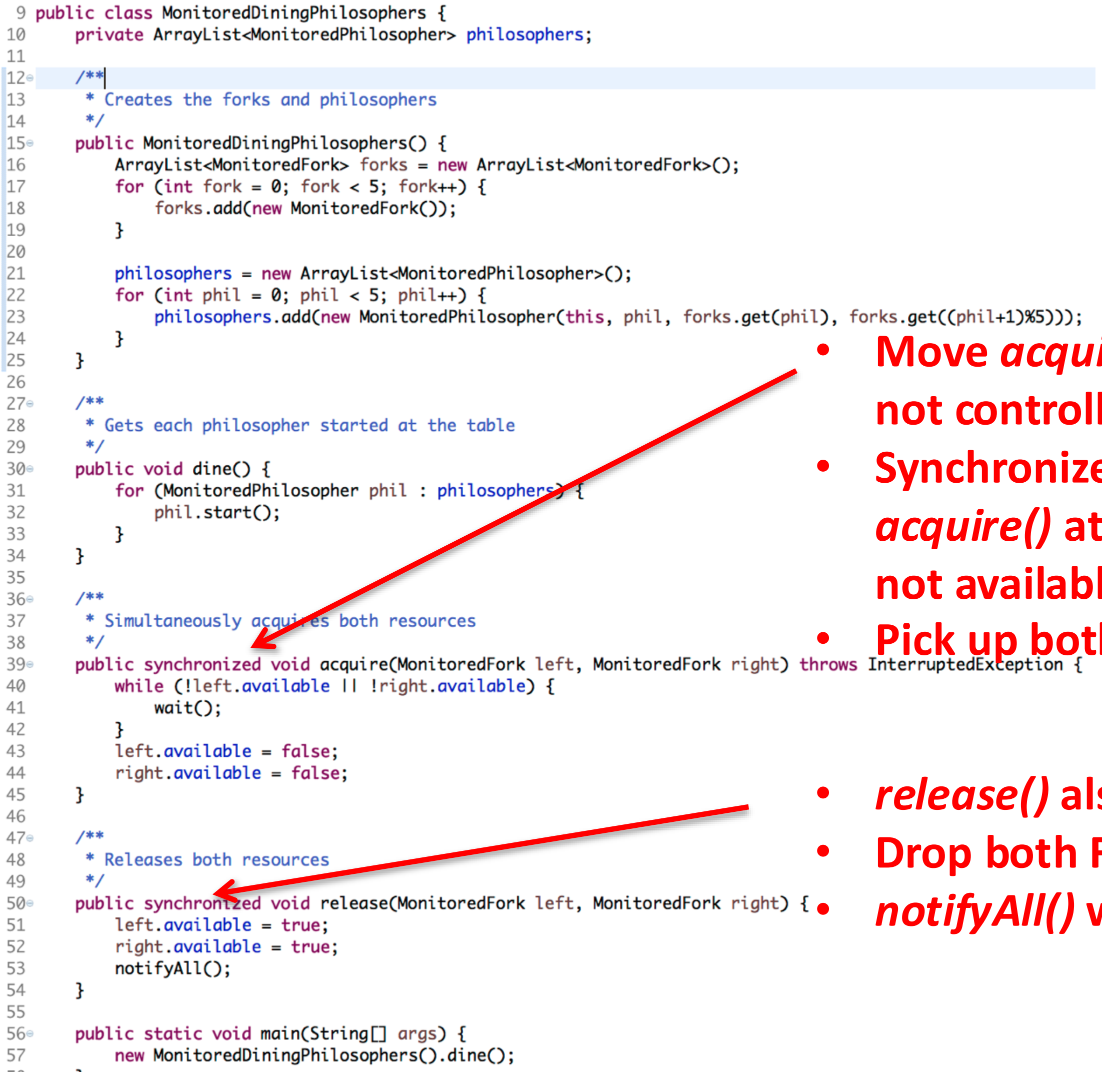


Eliminate hold and wait

- Make picking up both forks an atomic operation
- Forks no longer control their destiny as in prior code
- Now we lock both with a mutex
- Could lead to *starvation* if one philosopher always picks up before another
- In this case starvation will eventually end because the philosophers only eat a limited number of meals

Prevent deadlocks by making getting both Forks an atomic operation

MonitoredDiningPhilosopher.java



```java
public class MonitoredDiningPhilosophers {
    private ArrayList<MonitoredPhilosopher> philosophers;

    /**
     * Creates the forks and philosophers
     */
    public MonitoredDiningPhilosophers() {
        ArrayList<MonitoredFork> forks = new ArrayList<MonitoredFork>();
        for (int fork = 0; fork < 5; fork++) {
            forks.add(new MonitoredFork());
        }

        philosophers = new ArrayList<MonitoredPhilosopher>();
        for (int phil = 0; phil < 5; phil++) {
            philosophers.add(new MonitoredPhilosopher(this, phil, forks.get(phil), forks.get((phil+1)%5)));
        }
    }

    /**
     * Gets each philosopher started at the table
     */
    public void dine() {
        for (MonitoredPhilosopher phil : philosophers) {
            phil.start();
        }
    }

    /**
     * Simultaneously acquires both resources
     */
    public synchronized void acquire(MonitoredFork left, MonitoredFork right) throws InterruptedException {
        while (!left.available || !right.available) {
            wait();
        }
        left.available = false;
        right.available = false;
    }

    /**
     * Releases both resources
     */
    public synchronized void release(MonitoredFork left, MonitoredFork right) {
        left.available = true;
        right.available = true;
        notifyAll();
    }

    public static void main(String[] args) {
        new MonitoredDiningPhilosophers().dine();
```

- Move *acquire()* and *release()* to main program, not controlled by individual Forks now
- Synchronized only allows one Philosopher in *acquire()* at a time, wait if left and right Forks not available
- Pick up both Forks while here

- *release()* also synchronized
- Drop both Forks while here
- *notifyAll()* when Forks are available