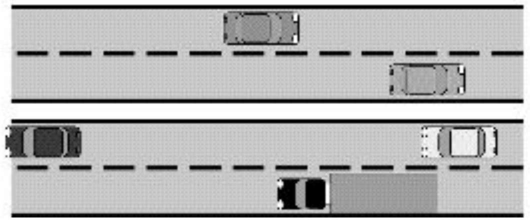
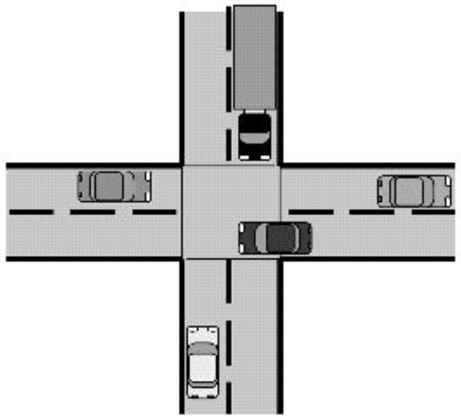
Concurrency in Java

Concurrency in General

- Multiple events happening at the same time.



- The trouble (in real life and in software development) is the interaction between these events.



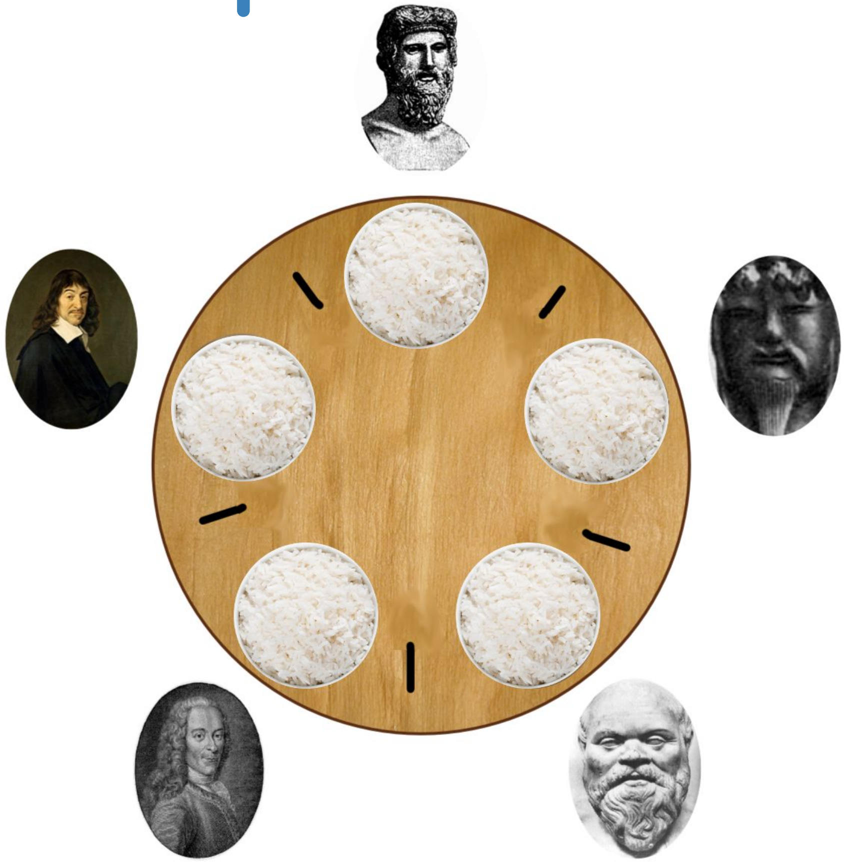
Lecture(s) Outline

- Review common concurrency examples
- Define concurrent principles / terms
- Rephrase common concurrency examples with the defined terms
- Introduction to concurrency specific to the Java programming language.
- Discuss Java programming language threading mechanisms

Dining Philosophers Problem

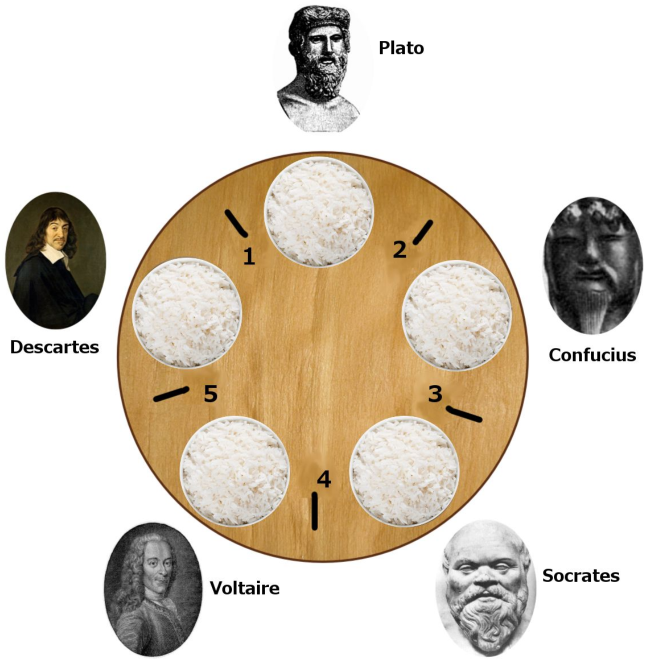
- Five silent philosophers sit around a table. In front of each is a bowl of rice. To each side of the philosopher is a single chopstick. Each philosopher must alternatively eat and think. A philosopher can pick up a chopstick on either side (right or left) but can only begin eating when holding two chopsticks. After eating the philosopher places both chopsticks back down. Assume the bowl of rice is bottomless (infinite rice within it).
- Problem - devise a scheme such that no philosopher starves.
  - Easy way to starve everyone is to have them all start by picking up the chopstick to their left.

Dining Philosophers Problem (cont)



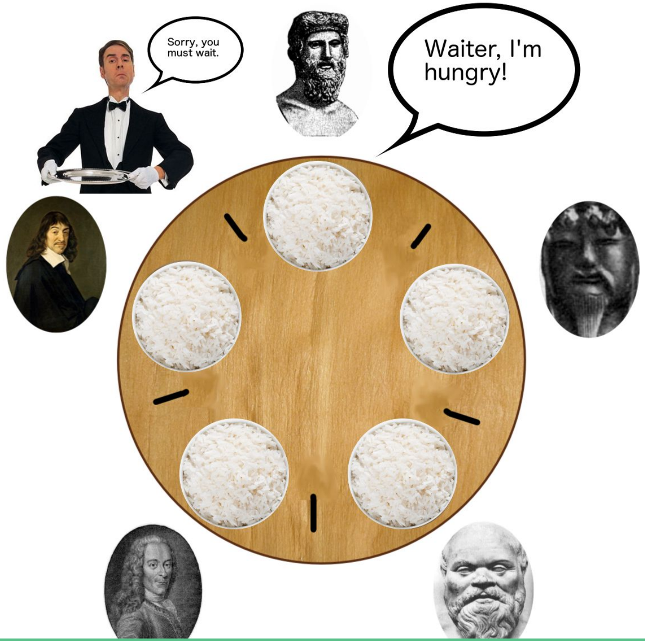
Dining Philosophers Problem (cont)

- Dijkstra’s Solution (he also devised the problem!)
    - Number the chopsticks. The philosopher must always first pick up the lower of the two chopsticks around him, then the higher.



Dining Philosophers Problem (cont)

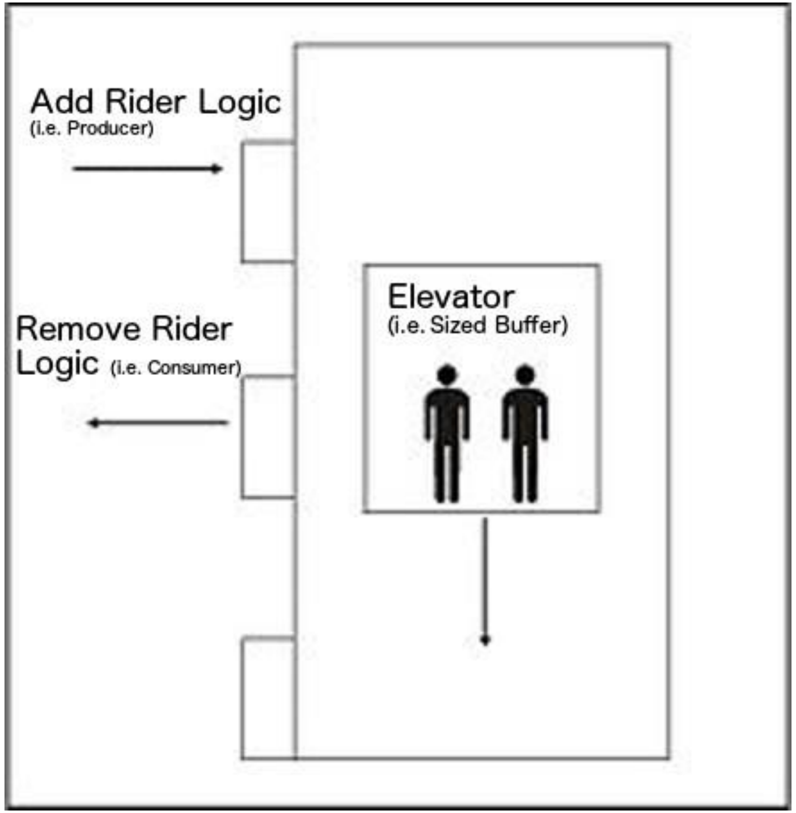
- Arbitrator “Waiter” Solution
  - Philosophers must ask the waiter for permission to pick up a chopstick and the waiter only gives permission to one philosopher at a time.



Producer / Consumer Problem

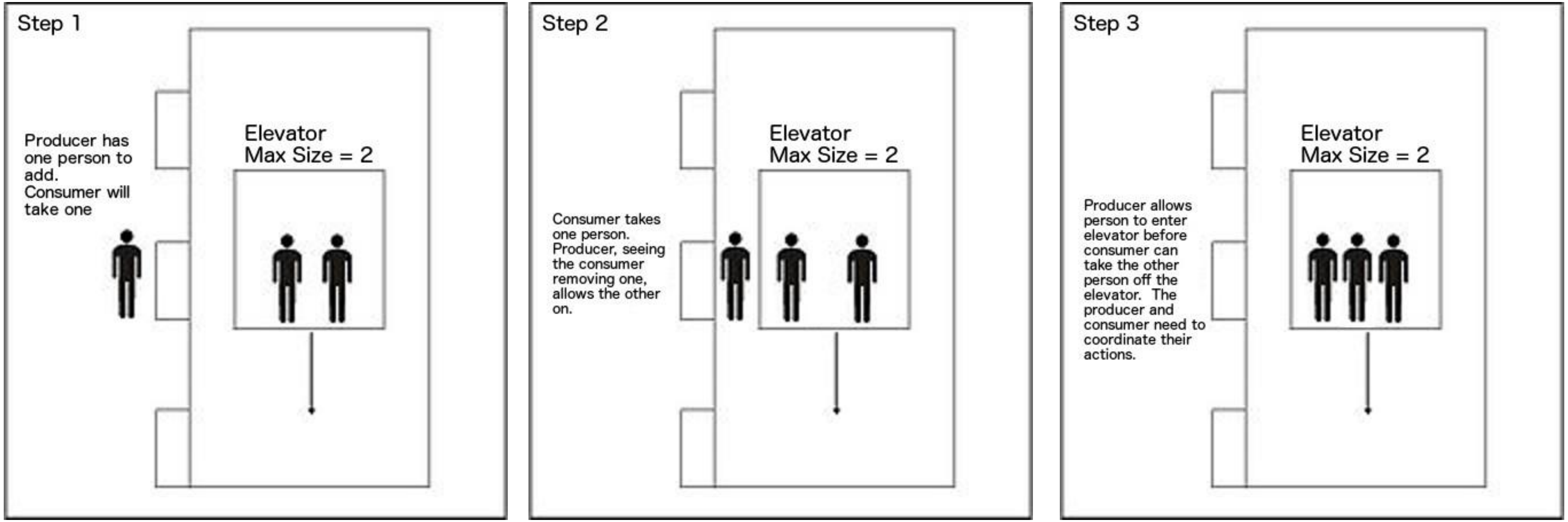
- There's a producer and a consumer sharing a fixed sized buffer. The job of the producer is to create data to put into the buffer. The job of the consumer is to remove data from the buffer.
  - The problem is to ensure the producer does not put data into a full buffer and the consumer does not try to remove data from an empty buffer.
- A real life example of this might be an elevator. The elevator itself is the buffer as it's fixed size (there is a maximum amount of people, or weight, allowed before it breaks). The producer is the logic responsible for allowing people onto the elevator. The consumer is the logic responsible for allowing people off.

Producer / Consumer Problem (cont)



Producer / Consumer Problem (cont)

- Problem can arise for the elevator if the same floor has a producer and a consumer and they do not coordinate their movements.


Step 1
Producer has one person to add. Consumer will take one
Elevator Max Size = 2
Step 2
Consumer takes one person. Producer, seeing the consumer removing one, allows the other on.
Elevator Max Size = 2
Step 3
Producer allows person to enter elevator before consumer can take the other person off the elevator. The producer and consumer need to coordinate their actions.
Elevator Max Size = 2

Concurrency Terminology

- Race condition
- Deadlock
- Resource Starvation
  - Livelock
- Atomicity
- Mutual Exclusion
- Locks / Semaphores
  - Binary Semaphore
  - Mutexes
- Monitors

Race Condition

- When output is dependent upon the uncontrollable sequence of events.
  - Becomes a bug if the sequence of events happens in a way the programmer did not intend.
  - Means that a program can behave “correctly” even though it suffers from a race condition.
- The producer-consumer elevator problem suffers from a race condition. The output is dependent upon the order of the persons entering and leaving the elevator. In some cases it works (no bug), in some cases it fails (bug) but in all cases it suffers from a race condition.
- Example!

Deadlock

- Two (or more) resources are waiting for each other to finish and thus neither ever do finish.
  - We saw this in the Dining Philosophers. If we make a scheme where all philosophers pick up left chopstick and then wait to pick up right chopstick before putting down the left.
- Example!

Resource Starvation

- When a process/thread is perpetually denied resources.
  - An example of this is livelock. This is different than deadlock in that the program isn’t halted it however cannot move forward because it is continuously denied a resource.
  - In the Dining Philosophers problem change the deadlock scheme such that after all picking up the left chopstick the philosophers wait ten minutes, but down the left chopstick and then all pick up the right chopstick, wait ten minutes, etc.
- Example!

Atomicity

- An operation (or set of operations) appears to occur to components of a system instantaneously.
    - From Java Concurrency in Practice (2.2.3, page 22): “Operations A and B are atomic with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either all of B has executed or none of it has.”
- The race condition in the producer-consumer-elevator problem and in the example were an indirect result of the assumption that a set of operations were atomic.
    - In the case of the elevator, that the count of the persons in the elevator and the act of removing the person was atomic.
    - In the case of the example, the retrieval and addition of the unique counter.
- Example!

Atomicity (cont)

- Operations on `long` and `double` are not atomic (unless you use the `volatile` keyword- more on that later)
    - https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.7

Mutual Exclusion

- For concurrent programs, mutual exclusion ensures that no two concurrent processes/threads are in a critical section at the same time.
    - A critical section is any portion of code which accesses a shared resource which cannot be accessed concurrently.
- E.g., in the elevator example, there needs to be a mutual exclusion guarantee around the consumer decrementing the count of persons within the elevator and the act of removing a person from the elevator.
- E.g., in the ‘UniqueNumberGenerator’ example, there needs to be mutual exclusion around returning a unique value and incrementing to the next value.

Locks & Semaphores

- Locks - a synchronization mechanism to limit access to a shared resource; a way of implementing mutual exclusion.
- Semaphores - a particular type of Lock.
  - Records how many units of a resource are available in conjunction with providing a safe (prevents race conditions) means of adjusting the record and potentially waiting for resources to become available.
    - A semaphore with an arbitrary resource count is called a **counting semaphore**.
    - A semaphore with value either 0 or 1 is called a **binary semaphore**.
  - **Mutex** - similar to a binary semaphore but often associates the notion of an owner such that only the owner can unlock the mutex (increment the semaphore back to 1).
- Example!

Next Lecture Reading

Java Concurrency in Practice; Brian Goetz et al. ISBN-13 978-0321349606 - Chapters 5 - 7

Homework 9

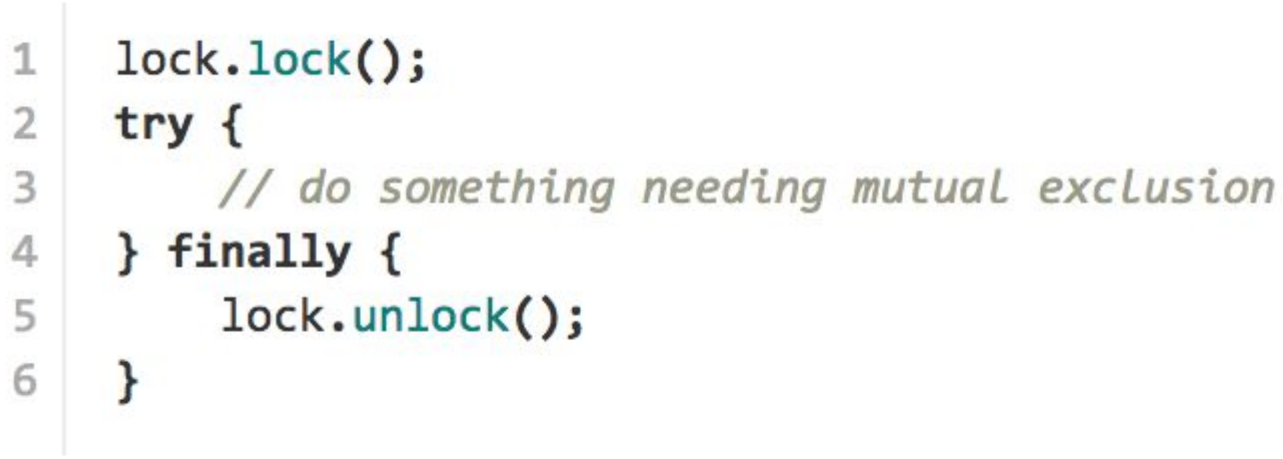
https://github.com/NYU-CS9053/Spring-2019/homework/week9

Monitor

- A synchronization mechanism that allows threads to have mutual exclusion and the ability to wait (block) for a condition to become true (e.g., there's room within the elevator). Monitors have a mechanism to signal waiting threads that a condition may have been met.
  - Monitors are implemented with a mutex lock and condition variables.
- Java loosely based its means of synchronization of objects based on this concept. However, fields are not all private (exposing state), methods are not forced to be synchronized (mutual exclusion) and the intrinsic lock (mutex) is exposed (it's the object itself).
- Example!

Monitors in Java

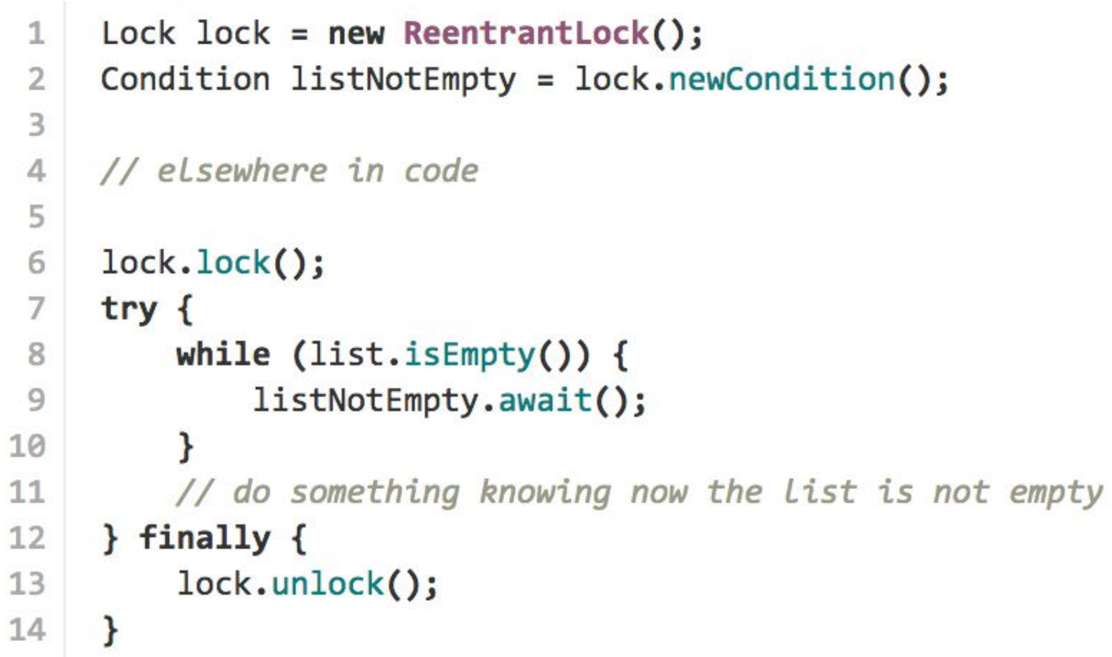
- Since Java 1.5, there is an explicit Lock/Condition object to give semantics of a monitor.
- General idea is to have one Lock object for all threads (per logical grouping of code needing mutual exclusion).



```java
lock.lock();
try {
    // do something needing mutual exclusion
} finally {
    lock.unlock();
}
```

Monitors in Java (cont)

- If the code within the mutual exclusion block should only be executed based on the state of other values use a Condition.



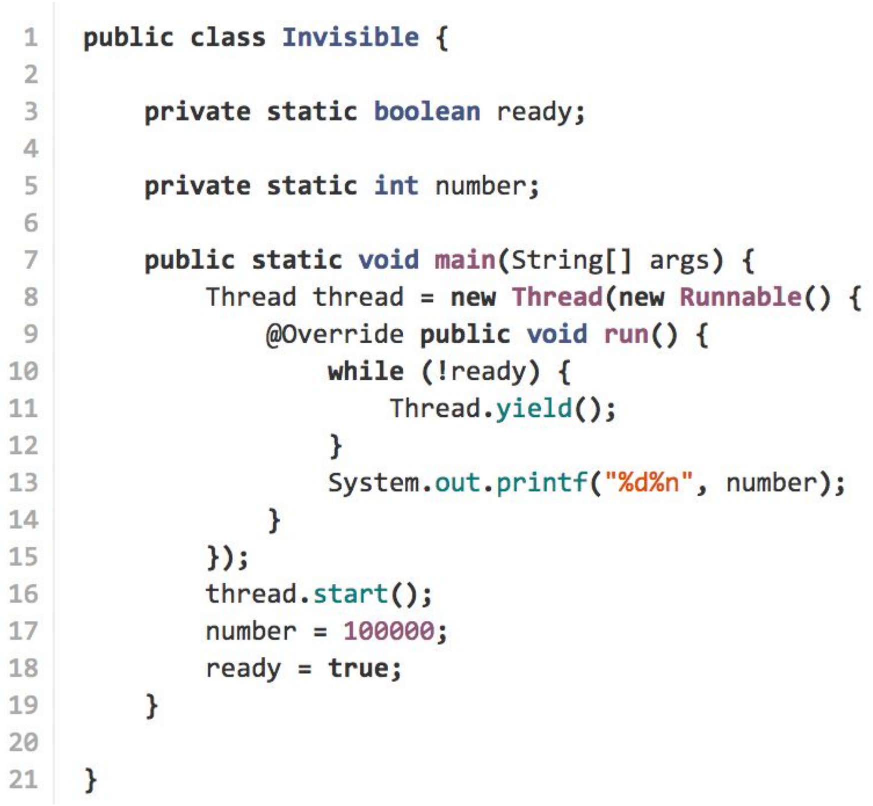
```java
Lock lock = new ReentrantLock();
Condition listNotEmpty = lock.newCondition();

// elsewhere in code

lock.lock();
try {
    while (list.isEmpty()) {
        listNotEmpty.await();
    }
    // do something knowing now the list is not empty
} finally {
    lock.unlock();
}
```

Threads in Java

- How?
  - By default there is one main Thread.
    - To date, all of your code written in Java has been run on this thread.
  - You can create additional Thread objects and start them.
  - Key elements are the Runnable object and the start and join methods.
  - Never call `run` or `stop`
  - Careful about interruptions -> always handle the, rethrow or terminate. If not handling them, call `Thread.currentThread().interrupt()`
- Example!

Visibility

- What happens when this program runs?
  - A) Prints 100000
  - B) Prints 0
  - C) Prints nothing
  - D) Any of the above



```java
public class Invisible {

    private static boolean ready;

    private static int number;

    public static void main(String[] args) {
        Thread thread = new Thread(new Runnable() {
            @Override public void run() {
                while (!ready) {
                    Thread.yield();
                }
                System.out.printf("%d%n", number);
            }
        });
        thread.start();
        number = 100000;
        ready = true;
    }

}
```

Modified example from Goetz's Java Concurrency in Practice (Listing 3.

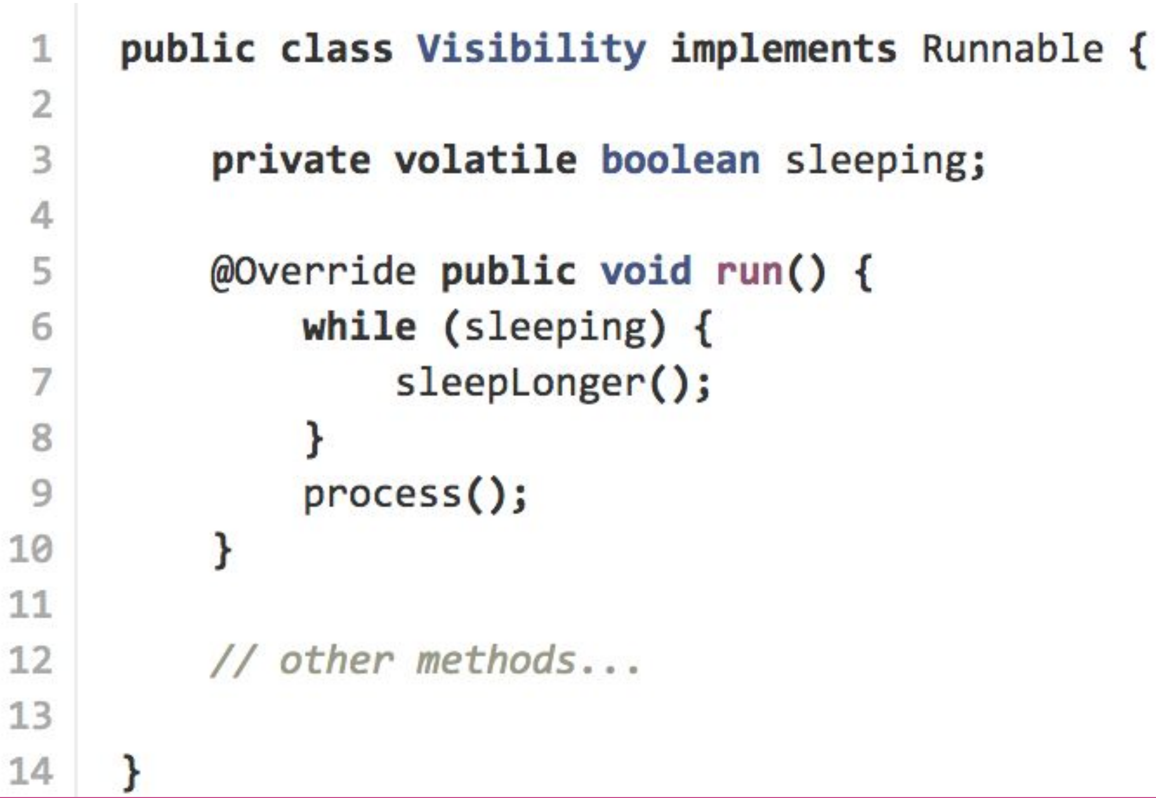
Visibility (cont)

- Writes to a shared variable on one thread can be seen by other threads.
    - With multiple threads, writing to a shared variable does not guarantee that other threads will immediately see the update (or ever see the update).
    - To ensure visibility synchronization techniques must be used.

The `volatile` Keyword

- Provides a weaker form of synchronization than using locking mechanisms.
- Marking a variable with `volatile` alerts the JVM that the variable will be shared amongst threads. Its value will not be cached (e.g. in registers); thus a read of a `volatile` variable will always be the most recent value written by a thread.
- CAREFUL! This does not make your variable/program thread-safe.
  - Good use of `volatile` is simply to ensure the value of a variable is seen by all threads.
  - E.g., the semantics of `volatile` are not strong enough to make `counter++` atomic (and thus thread safe).
    - Use the Atomic variants of Boolean, Integer, Long.

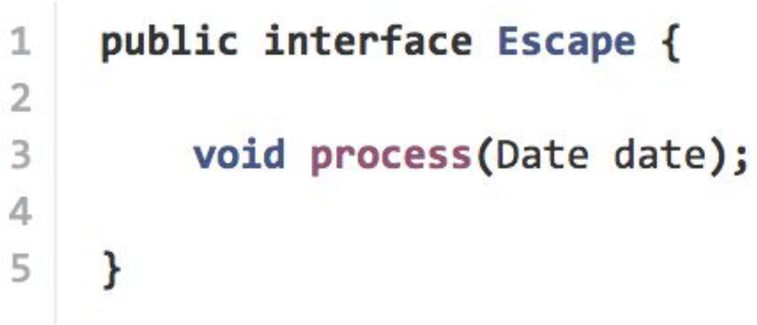
Example Using volatile



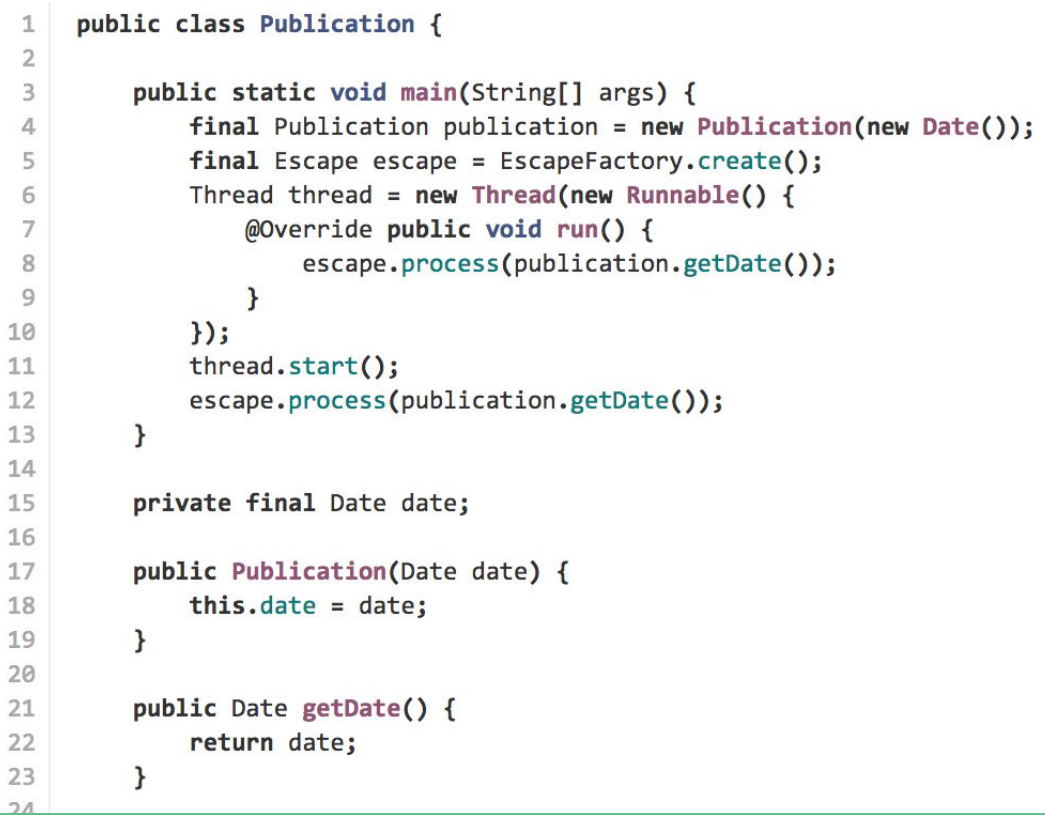
```java
public class Visibility implements Runnable {

    private volatile boolean sleeping;

    @Override public void run() {
        while (sleeping) {
            sleepLonger();
        }
        process();
    }

    // other methods...

}
```

Publication & Escape

- Is this program thread-safe?



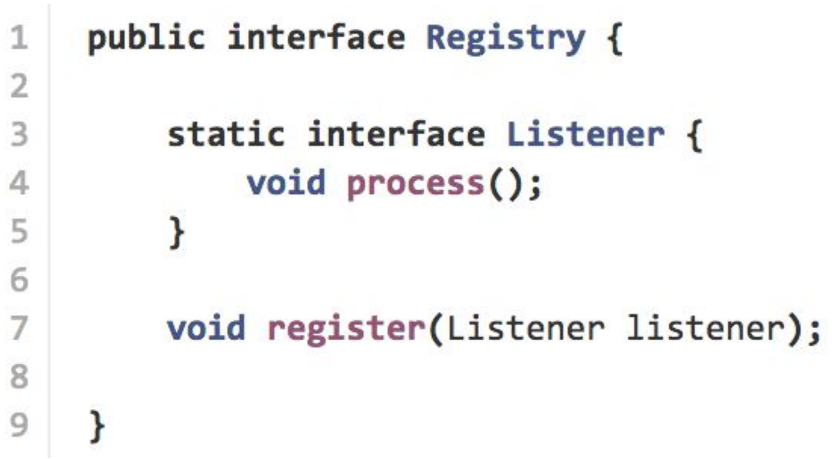
```java
public interface Escape {

    void process(Date date);

}
```



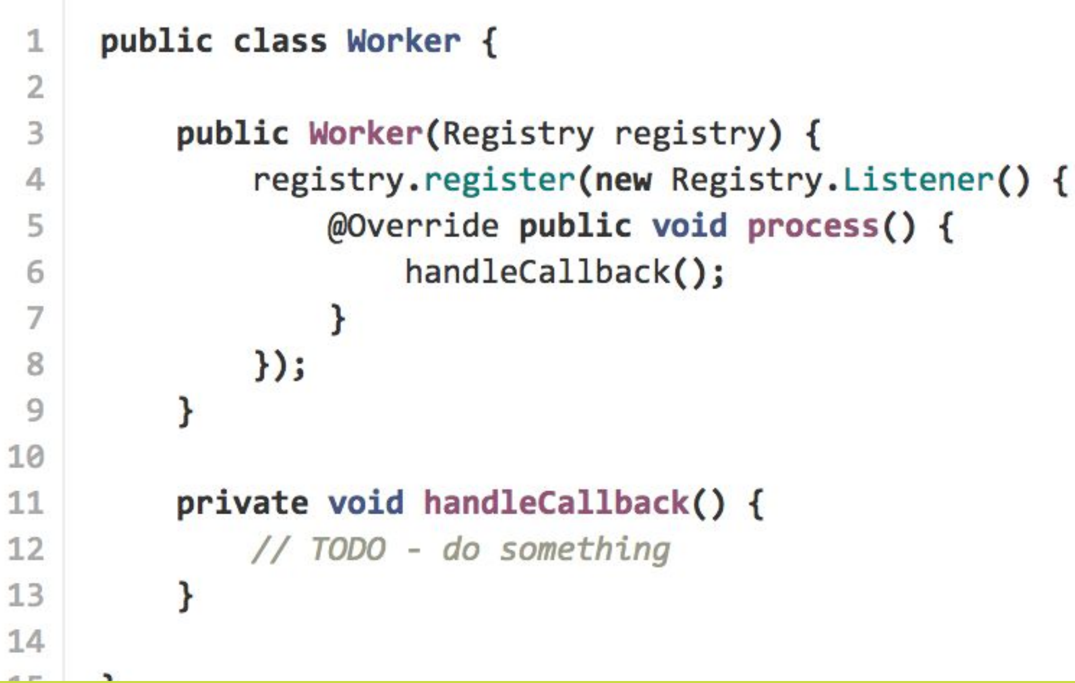
```java
public class Publication {

    public static void main(String[] args) {
        final Publication publication = new Publication(new Date());
        final Escape escape = EscapeFactory.create();
        Thread thread = new Thread(new Runnable() {
            @Override public void run() {
                escape.process(publication.getDate());
            }
        });
        thread.start();
        escape.process(publication.getDate());
    }

    private final Date date;

    public Publication(Date date) {
        this.date = date;
    }

    public Date getDate() {
        return date;
    }
```

Publication & Escape (cont)

- What does this program do incorrectly?



```java
public interface Registry {

    static interface Listener {
        void process();
    }

    void register(Listener listener);

}
```



```java
public class Worker {

    public Worker(Registry registry) {
        registry.register(new Registry.Listener() {
            @Override public void process() {
                handleCallback();
            }
        });
    }

    private void handleCallback() {
        // TODO - do something
    }

}
```

Immutability

- “Immutable objects are always thread-safe” - Goetz’s Java Concurrency in Practice, 3.4
- Immutable definition (see Goetz 3.4); object is immutable if
  - All its fields are final
  - Its state cannot be modified after construction
  - It is properly constructed (does not escape during construction)
- For this (and all the other reasons prior to concurrency we mentioned in lectures) prefer immutable objects. Only use mutable objects if there’s a compelling reason not to

Synchronization Mechanisms

- Revisit Lock/Condition
  - Lock in Java - ReentrantLock
  - Lock with Condition
  - Lock without Condition
    - `edu.nyu.cs9053.concurrency.chapter14.WithCondition`
      `edu.nyu.cs9053.concurrency.chapter14.WithoutCondition`
  - IMPORTANT - Must own lock to await on condition and must own lock to signal a condition
    - `edu.nyu.cs9053.concurrency.chapter14.UnownedLockConditionInvoker`

Java's Built-in Synchronization

- Java loosely based its means of synchronization of objects based on this concept. However, fields are not all private (exposing state), methods are not forced to be synchronized (mutual exclusion) and the intrinsic lock (mutex) is exposed (it's the object itself).
- Every Object has an intrinsic lock (and one condition).
  - Not the same as Lock/Condition (these came in Java 1.5)
    - Every `synchronized` can be rewritten as Lock/Condition but not every Lock/Condition can be rewritten using `synchronized` (which is why Java 1.5 added Lock/Condition)
- Example!

Java synchronized (cont)

- Can you synchronized as a block or around any method.
  - If around method, using the intrinsic lock of the Object
    - So can you synchronize a static method?
      - What’s the Object for static methods?
  - If used within a block the programmer must specify which Object to use as the lock.
    - Can use this for the current Object
    - Can use Class.class for the Object’s class (to lock all Objects of a given class).
- Example!

Thread-safe Collections/Utilities

- Look at classes within package `java.util.concurrent`
- AtomicInteger, AtomicBoolean, AtomicLong, AtomicReference
  - Provides atomic interactions (get and set, etc)
- ConcurrentMap (interface) and ConcurrentHashMap (implementation)
  - Leverages “striped locking” - see http://www.ibm.com/developerworks/library/j-jtp07233/
- BlockingQueue - allows you to easily implement Producer/Consumer problems
- CountDownLatch
  - Allows work to be done until x values reach the gate
- CyclicBarrier
  - Similar to a CountDownLatch but can be reused

Executors

- Java 1.5 provided a framework for interacting with Threads which abstracts logic of starting/stopping and executing work.
  - `Executor` - interface of something which can process work
  - `ScheduledExecutor` - extends Executor but allows the programmer to control over when and how frequent work is done
  - `ExecutorService` - similar to a “thread pool”; owns a number of Executor objects. Get an instance of this and give it work.
  - `ScheduledExecutorService` - same as an ExecutorService but also can schedule jobs (ScheduledExecutor).
  - `Executors` - utility class which creates ExecutorService and ScheduledExecutorService objects.
- Example!

Homework 10

https://github.com/NYU-CS9053/Spring-2019/homework/week10