



# Threads

Ch 9

# Motivation

- Create GUI for program which finds primes
  - Using very slow algorithm:  
~20 seconds to find a prime.
  - Want UI to be responsive while computing primes.
- Demo: ThreadDemoUI.java (ca.threads.primeui)
  - 1) Single threaded:.. UI locks up.
  - 2) Background thread:.. UI responsive.
  - 3) Many threads:.. Use all cores.

# Topics

- 1) How can our program do 2 things at once?
- 2) Does doing 2 things at once cause problems?



# Thread Basics: Runnable & Thread

# Running Task

## 1) Create a Task...Class which does the work.

Must implement  
**Runnable:**

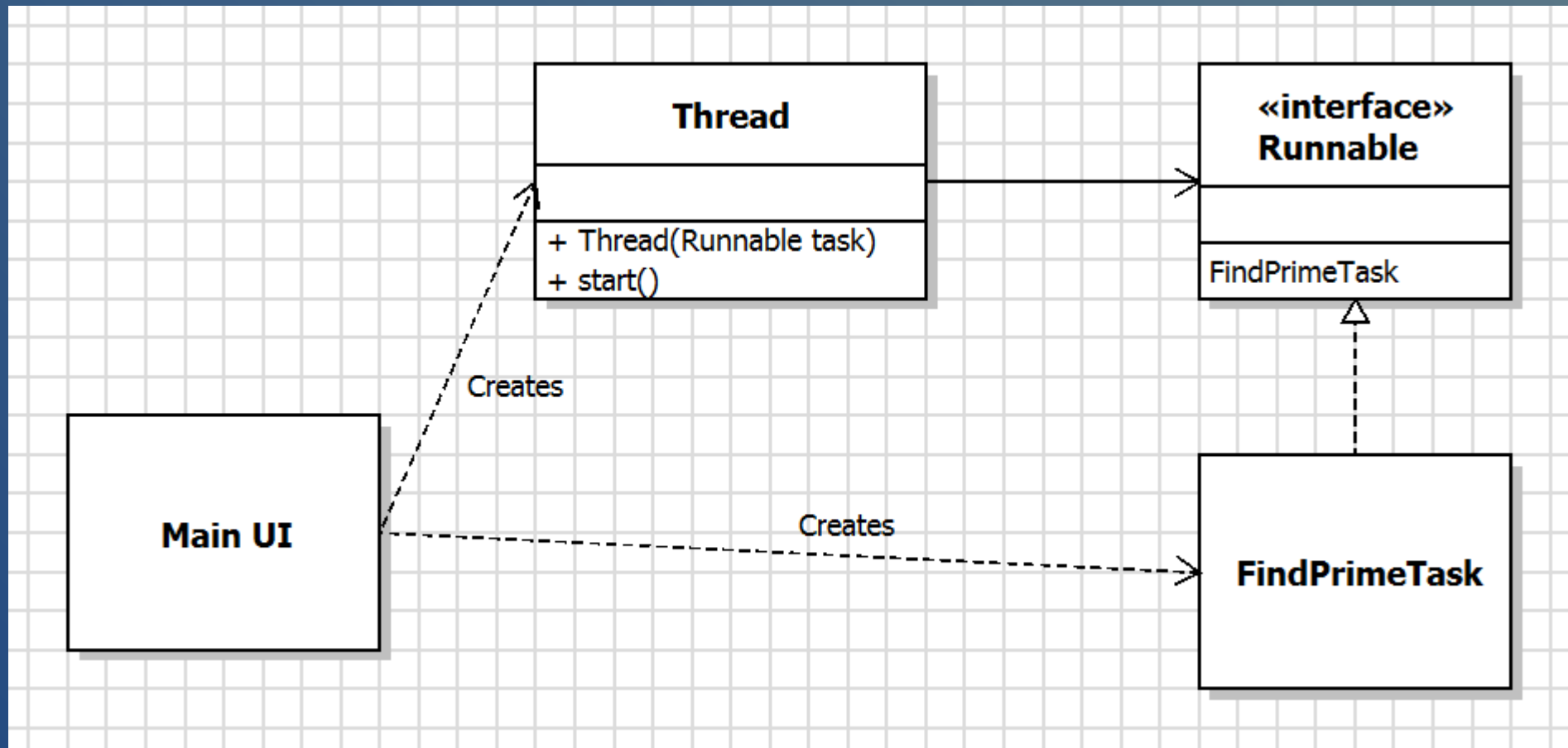
```
public interface Runnable {  
    void run();  
}
```

```
class MyAmazingTask implements Runnable {  
    @Override  
    public void run() {  
        // Calculate something amazing here!  
    }  
}
```

## 2) Create a.. thread to execute the task

```
public void main(String[] args) {  
    Runnable myTask = new MyAmazingTask();  
    Thread myThread = new Thread(myTask);  
    myThread.start();  
}
```

# UML for Prime Demo



# Timing

- **Time Slice:**  
a block of time during which..  
**a thread can execute.**
  - OS/JVM allocates time-slices to threads.
- **Not always equal:**
  - **Starvation:** a task given.. **no time to execute**
  - **Fairness:** Often use round-robin scheduling.
  - **Priority:** Some threads higher priority than others.
- **UI Demo:**
  - 10 threads computing if same number is prime.  
Will not all.. **take same amount of time.**

# Suspending a Thread

- Can briefly suspend a thread with..  
`Thread.sleep(delay);`
  - delay is in milliseconds (1/1000 second)
  - can throw `InterruptedException`

```
private static final long DELAY_MS = 1000;

@Override
public void run() {
    try {
        while (true) {
            System.out.println("Hello!");
            Thread.sleep(DELAY_MS);
        }
    } catch (InterruptedException e) {
        // Handle end of task here.
    }
}
```



# Thread Synchronization



# Thread Interactions

- Race condition
  - Effect of multiple threads on shared data depends on.. order in which threads are scheduled.
  - Demo: MathDemo
- Cause
  - The execution of one thread is interrupted by another thread.
  - Second thread disturbs or corrupts operation of initial thread.
- Critical Section
  - A portion of a thread's execution where.. it can suffer a race condition.

# MathDemo Analysis

One possible scenario:

Thread 1:

number = 5

Preempted

number still 7

```
volatile private int number;
```

```
public int compute(int newValue) {  
    number = newValue;  
}
```

```
    int result = 0;  
    for (int i = 0; i < NUM_STEPS; i++) {  
        result += number;  
    }
```

```
    for (int i = 0; i < NUM_STEPS; i++) {  
        result -= number;  
    }
```

```
    return result;  
}
```

```
}
```

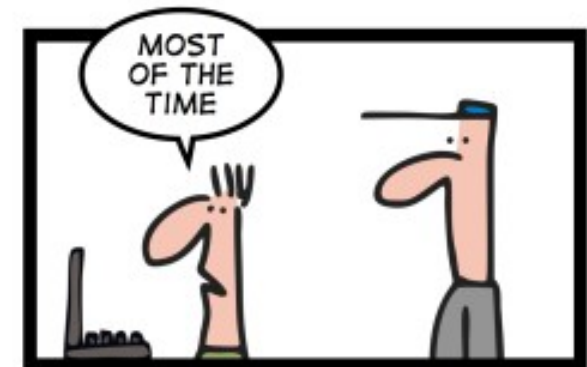
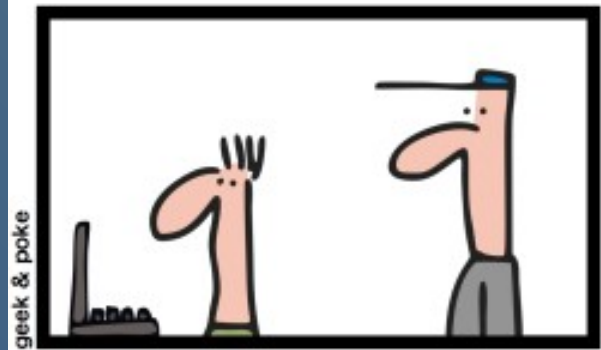
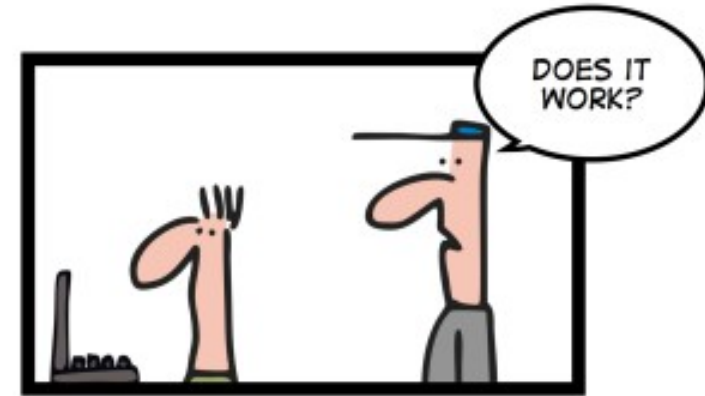
Thread 2:

number = 7

# Heisenbug

- **Race Condition Solution**
  - **Thread Safe**: No race conditions.
  - How? Use locks.
- **Aside: Non-reproducible bugs**
  - Dependent on subtle timing events
  - **Heisenbug**: A bug who's behaviour is.. **changed by looking for it**.
  - Debugging can change thread timing, changing the behaviour.
  - VERY tricky bugs to find!

SIMPLY EXPLAINED



CONCURRENCY

# Locks

- **Process:**

1. **Create a lock** for access to some resource (such as a variable, file, printer, ...)
2. **Lock the lock** before accessing resource.
3. Use resource
- 4... **Unlock the lock when done.**

```
class LockExample {  
    private ReentrantLock myLock = new ReentrantLock()  
    public void foo() {  
        myLock.lock();  
        try {  
            // Protected critical section  
            // ... do stuff here  
        } finally {  
            myLock.unlock();  
        }  
    }  
}
```

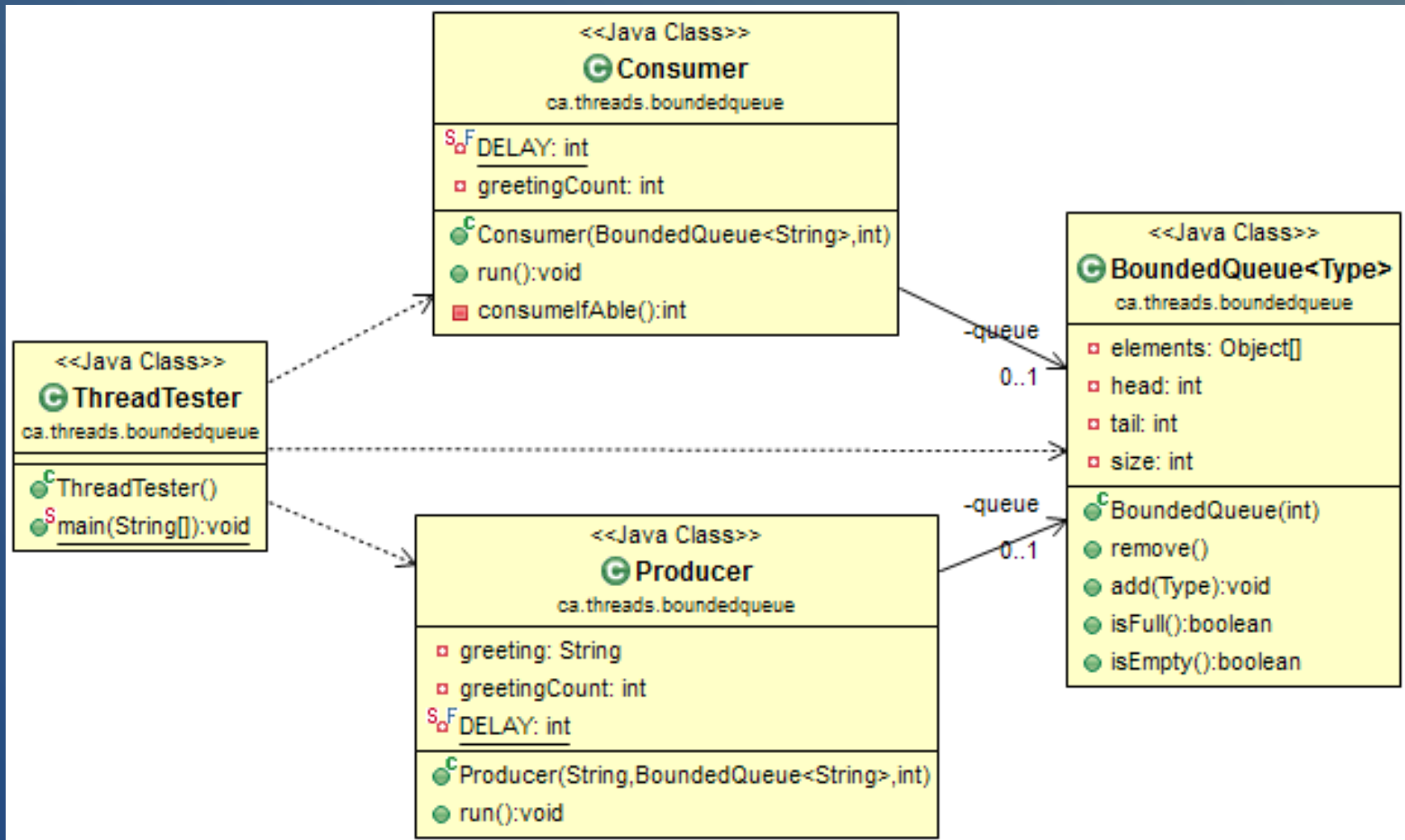
.. No other thread can execute this code while this thread has it locked.

# Locking Example

- Dealing with a shared queue.
  - Producers  
threads adding data to a bounded queue
    - Ex: calculating prime numbers.
  - Consumer  
thread removing data from a bounded queue
    - Ex: printing out the prime numbers.
- Thread Synchronization Problem
  - Two producers may interfere with each other.
  - Consumer and producer may interfere.
- Thread safe..  
Works correctly with multiple threads.

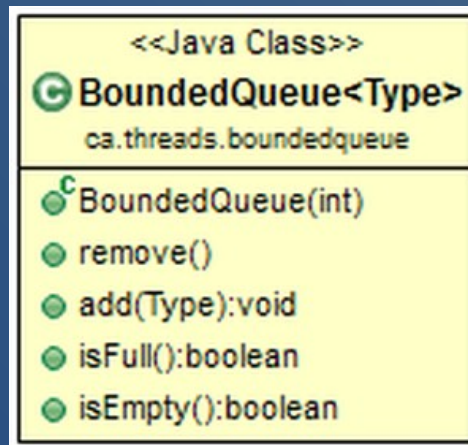


# Producer / Consumer UML



# Producer / Consumer

```
public class Producer implements Runnable {  
    // Passed the queue from main()  
    private BoundedQueue<String> queue;  
  
    public void run() {  
        while (..) {  
            if (!queue.isFull()) {  
                queue.add("Hello");  
            }  
            Thread.sleep(...);  
        }  
    }  
}
```

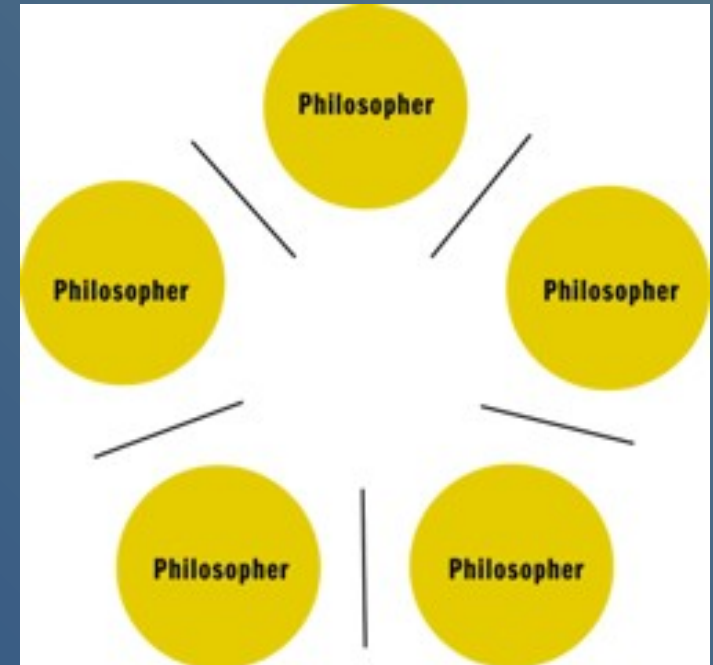


```
public class Consumer implements Runnable {  
    // Passed the queue from main()  
    private BoundedQueue<String> queue;  
  
    public void run() {  
        while (...) {  
            if (!queue.isEmpty()) {  
                String msg = queue.remove();  
                System.out.println(msg);  
            }  
            Thread.sleep(...);  
        }  
    }  
}
```

Note: Exception handling removed.

# Deadlock

- **Deadlock:**  
if no thread can proceed because..  
each is waiting for another to release a lock.
- **Ex: Dining Philosophers**
  - Philosophers are either:
    - Thinking or
    - Eating
  - To eat, a philosopher needs..  
two chopsticks.
  - How can deadlock happen?
  - How to resolve?



# Stopping a Thread

- Thread normally ends when.. **run() finishes**
- Can end a running thread (vs letting it finish):
  - *Notify* thread of interruption with:

```
Runnable myTask = new MyAmazingTask();
Thread myThread = new Thread(myTask);
myThread.start();

// ... Later, when thread not needed:
myThread.interrupt();
```
  - Interrupted thread knows it's interrupted by:
    - If in a **Thread.sleep()**, it throws exception.
    - Manually check the interrupted flag:

```
if (Thread.currentThread().isInterrupted()) {...}
```

# Summary

- **Process**
  - Create a **task**: Implement **Runnable**
  - Create a **thread**: pass it a runnable, call **start()**
  - Interrupt with **myThread.interrupt()**
- **Race Condition**: Threads may interfere
  - Solution: **locks**
- **Common Examples**
  - **Produce/Consumer**
  - **Dining Philosophers**
    - **Deadlocks**: Threads waiting on each-other.