Java Programming 2
Higher-level concurrency

Mary Ellen Foster

MaryEllen.Foster@glasgow.ac.uk

“Contemplating orange juice” by Hannah Webb
https://flic.kr/p/bkqXSy

Higher-level concurrency

Lock objects

Atomic variables

Concurrent collections

Synchronized methods: reminder

Additional keyword: **`synchronized`**

Add to method header

Ensures that:

- Two calls to **synchronized** methods **on the same object** cannot interleave
- When a synchronized method exits, it **happens-before** any other **synchronized** method calls **on the same object**

A synchronized method makes use of the **intrinsic lock** of the object

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

Lock objects

Generalised version of **synchronized** code (simple intrinsic lock)

Basic interface: **java.util.concurrent.locks.Lock**

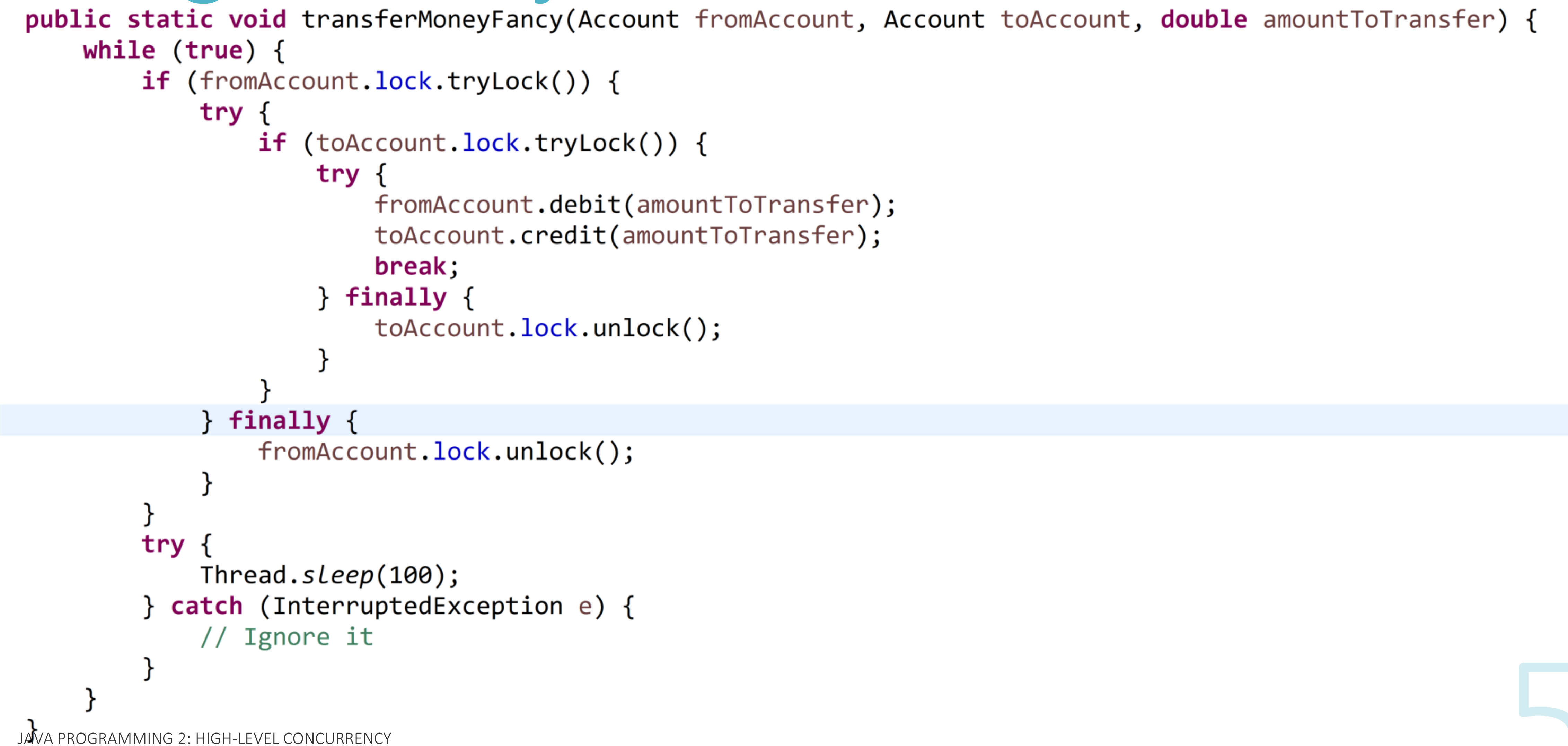
- Work like intrinsic locks
- Only one thread can own a Lock object at a time

Big advantage: allow code to back out of an attempt to acquire a lock

- **tryLock()** – backs out if lock is not available or if timeout expires (timeout is optional)
- **lockInterruptibly()** – backs out if another thread sends interrupt before lock is acquired

Best practice: put all code in a **try** block and call **lock.unlock()** in a **finally** clause

Using Lock objects to deal with deadlock



```
public static void transferMoneyFancy(Account fromAccount, Account toAccount, double amountToTransfer) {
    while (true) {
        if (fromAccount.lock.tryLock()) {
            try {
                if (toAccount.lock.tryLock()) {
                    try {
                        fromAccount.debit(amountToTransfer);
                        toAccount.credit(amountToTransfer);
                        break;
                    } finally {
                        toAccount.lock.unlock();
                    }
                }
            } finally {
                fromAccount.lock.unlock();
            }
        }
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            // Ignore it
        }
    }
}
```

Locks and Conditions

java.util.concurrent.locks.Condition – allows a thread to wait (not using any resources) until some condition is satisfied

Get a Condition object from a Lock object with **Lock.newCondition()**

Basic process:

- Acquire the lock on the object (lock(), tryLock(), lockInterruptibly(), etc)
- If thread needs to wait, call **condition.await()**
- When program is ready to continue, other thread calls **condition.signal()**
- Do whatever processing is needed …
- After all that, then call **lock.unlock()**

A single Lock can have multiple Condition objects to control different aspects

Atomic variables

Package **java.util.concurrent.atomic**

Defines classes that support **atomic operations** on single variables

- All classes have get() / set() methods that impose **happens-before** – set happens before get
- Atomic compareAndSet() method
- Simple arithmetic methods that apply to integer atomic variables
  - *decrementAndGet(), addAndGet(), getAndAdd() …*

Counters revisited

```java
class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {
        c++;
    }
    public synchronized void decrement() {
        c--;
    }
    public synchronized int value() {
        return c;
    }
}
```

```java
import java.util.concurrent.atomic.AtomicInteger;
class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);
    public void increment() {
        c.incrementAndGet();
    }
    public void decrement() {
        c.decrementAndGet();
    }
    public int value() {
        return c.get();
    }
}
```

Other useful Java libraries related to concurrent programming

- **java.util.concurrent:** Concurrent collections
  - **BlockingQueue**: a first-in/first-out structure that blocks when you attempt to add to a full queue or remove from an empty queue
  - **ConcurrentMap:** defines atomic operations on maps (e.g., **putIfAbsent**)
  - **ConcurrentNavigableMap:** supports approximate matches
- Streams (coming up soom) support **parallelStream()** operator – processes Stream objects in parallel (Java runtime decides how to divide things up)
  - Note that any methods called in the context of a parallel stream must be thread-safe (locks, atomic, etc)