



AARHUS UNIVERSITET

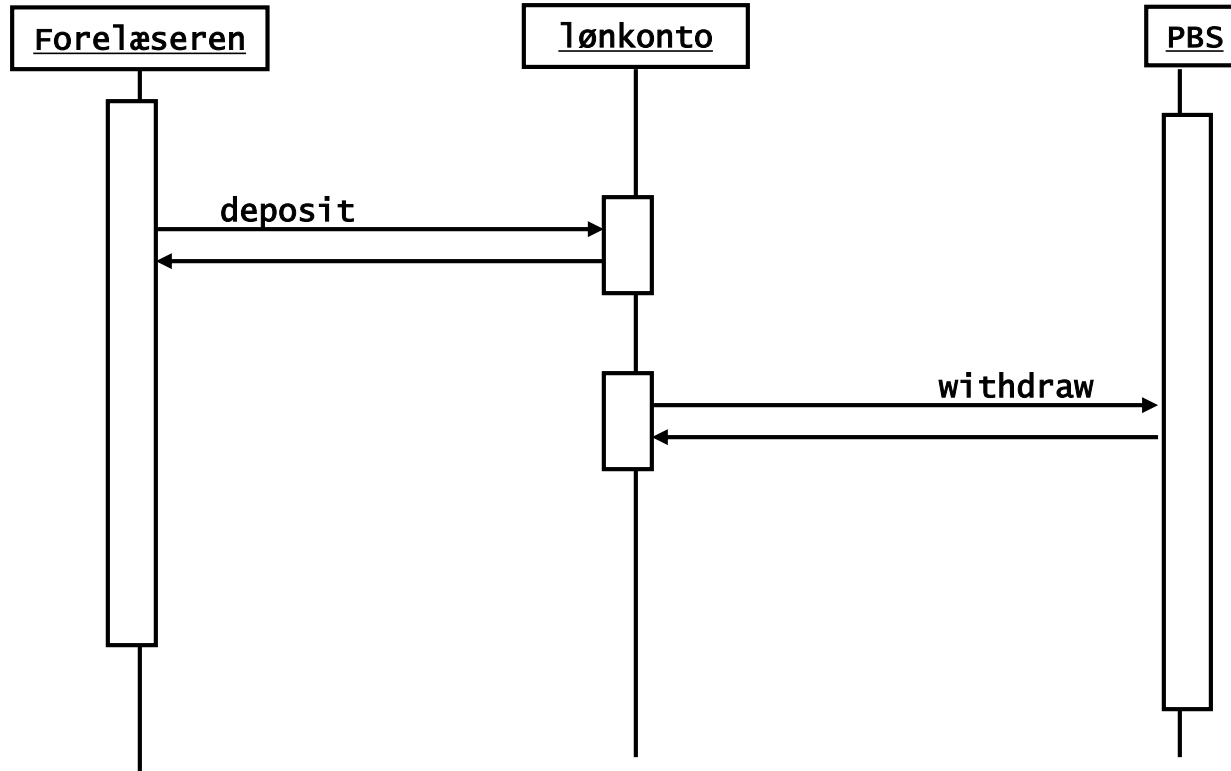
Software Engineering and Architecture

Concurrency

Shared Resources



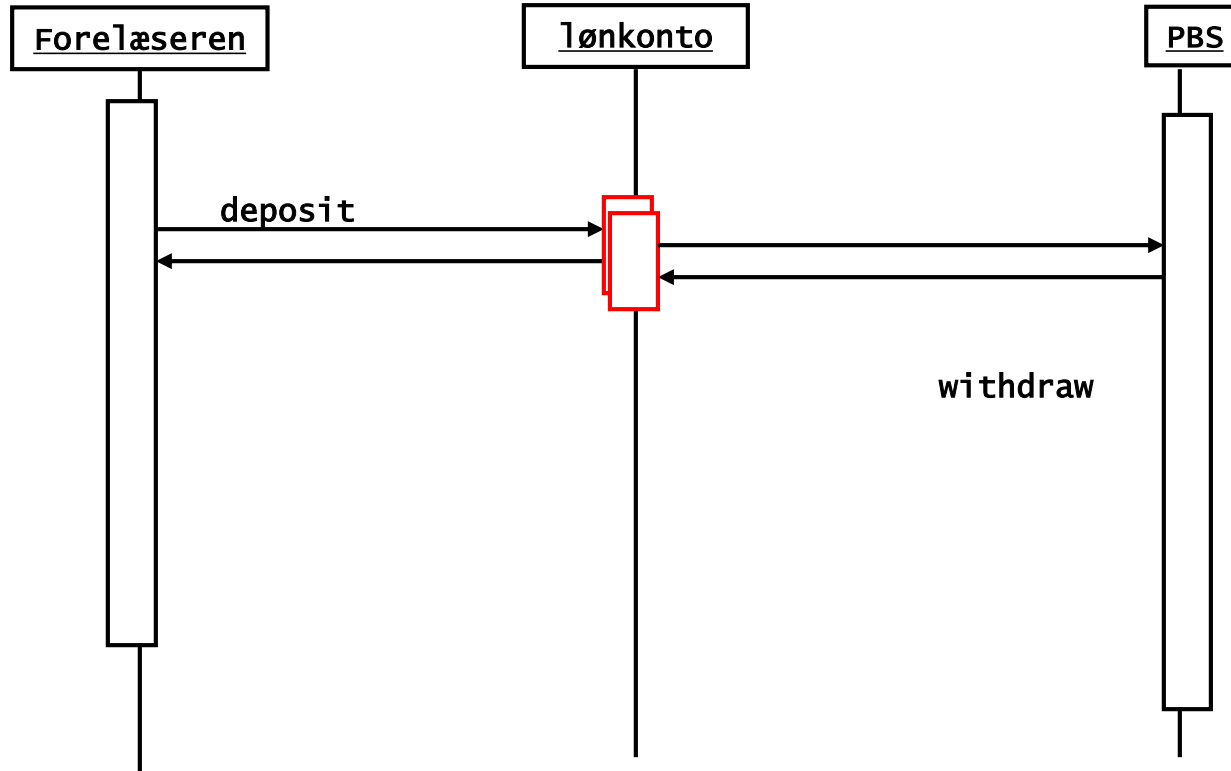
Two Threads – One Account



OK



Two Threads – One Account



Not OK



Example

- I insert 200kr while PBS withdraws 100kr. The balance is initially 50kr.

```
public boolean deposit(long amount){
    if(amount >= 0){
        return setBalance(getBalance()+amount);
    }
    else {
        return false;
    }
}
```

```
public boolean withdraw(long amount){
    if(amount >= 0){
        return setBalance(getBalance()-amount);
    }
    else {
        return false;
    }
}
```



Concurrent Execution

AARHUS UNIVERSITET

```

if(amount >= 0){
    return setBalance(getBalance()+200

```

Scheduler thread switch!

```

if(amount >= 0){
    return setBalance(getBalance()-100);
}
else{
    return false;
}
}

```

```

}
else{
    return false;
}
}

```

setBalance(50+200

b=50

setBalance(50-100);

b=-50

);

b=250;





Example!

- The 'System.out' is a shared resource !

```
PS D:\work\teaching\SWEA-E17\code\lab\threads\thread-demo1> java ThreadDemo
aaaaaaaaaaaaaaaaaaaaaaaaabaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbaabaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
PS D:\work\teaching\SWEA-E17\code\lab\threads\thread-demo1> java ThreadDemo
aaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbaabbbbbbbbbbbbbbbbbb
baaaabbaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
PS D:\work\teaching\SWEA-E17\code\lab\threads\thread-demo1> java ThreadDemo
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbaaaaaaaaaaaaaaaaaaaaaaaaaabbaabbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
PS D:\work\teaching\SWEA-E17\code\lab\threads\thread-demo1>
```



Example: Shared Counter

```
public class CounterTest {  
  
    public static void main(String[] args) throws InterruptedException {  
  
        final Counter counter = new Counter();  
  
        // create 1000 threads  
        ArrayList<MyThread> threads = new ArrayList<MyThread>();  
        for (int x = 0; x < 1000; x++) {  
            threads.add(new MyThread(counter));  
        }  
  
        // start all of the threads  
        Iterator i1 = threads.iterator();  
        while (i1.hasNext()) {  
            MyThread mt = (MyThread) i1.next();  
            mt.start();  
        }  
  
        // wait for all the threads to finish  
        Iterator i2 = threads.iterator();  
        while (i2.hasNext()) {  
            MyThread mt = (MyThread) i2.next();  
            mt.join();  
        }  
  
        System.out.println("Count : " + counter.getCount());  
        System.out.println("Expected: " + 1000 * 10000);  
    }  
}
```

```
class Counter {  
    private Integer count = new Integer(0);  
  
    public void incrementCount() {  
        count++;  
    }  
  
    public int getCount() {  
        return count;  
    }  
}
```

```
// thread that increments the counter 10.000 times.  
class MyThread extends Thread {  
    Counter counter;  
  
    MyThread(Counter counter){  
        this.counter = counter;  
    }  
  
    public void run() {  
        for (int x = 0; x < 10000; x++) {  
            counter.incrementCount();  
        }  
    }  
}
```

```
d:\proj\frsproject\concurrency>java CounterTest  
Count : 3492772  
Expected: 10000000
```

```
d:\proj\frsproject\concurrency>java CounterTest  
Count : 3547105  
Expected: 10000000
```

```
d:\proj\frsproject\concurrency>java CounterTest  
Count : 3062784  
Expected: 10000000
```

Source: <http://www.javacodex.com/Concurrency/AtomicInteger-Counter>



Race Condition

- Race Condition:
 - If multiple threads *writes* to resources then the outcome is determined by the sequence/timing in which events occur.
 - **BAD!** You have no control of the behavior of your program; and the result is erroneous
 - Inserting 200 and withdrawing 100 on balance 50 must be **150**
 - Our particular execution gave **250**
 - The next one may give **-50**
-
- Critical Section:
 - The code section in which race conditions can occur



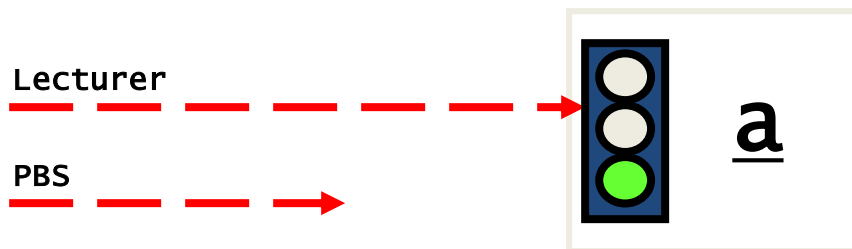
The Solution

AARHUS UNIVERSITET

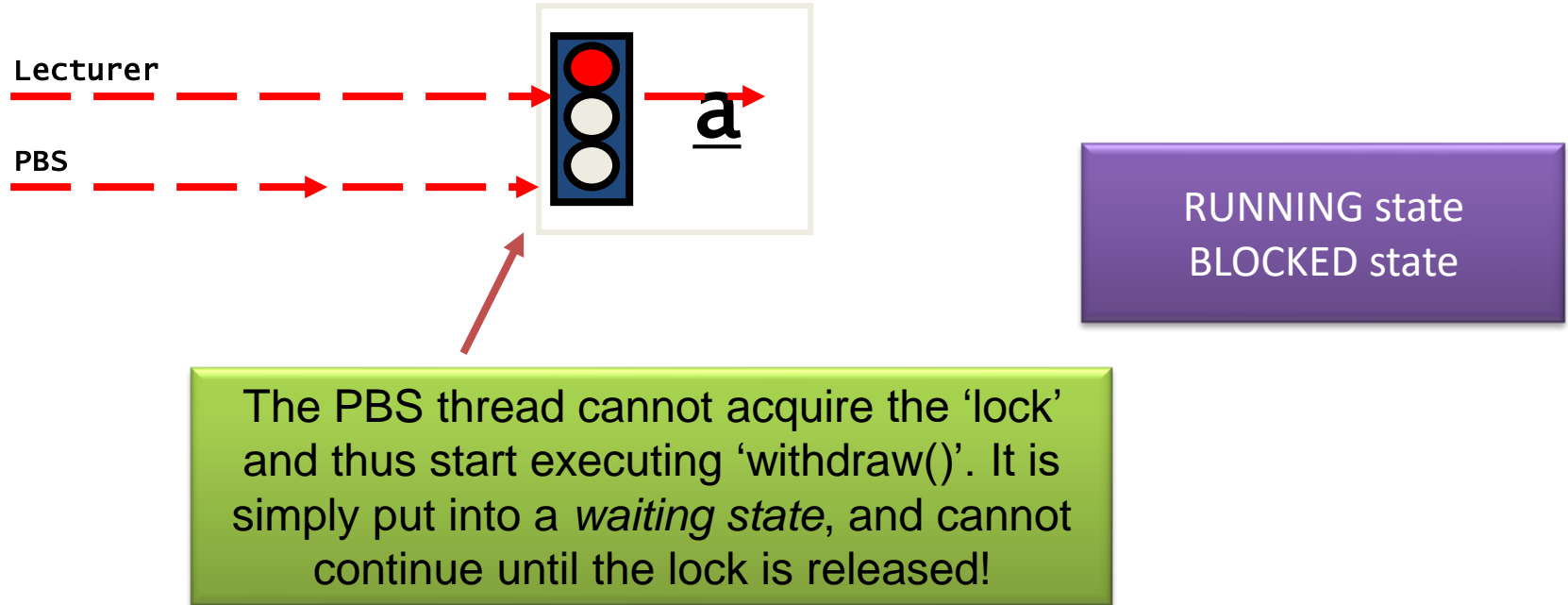
- *Critical sections must be treated as an atomic instruction*
 - *Da: "Udelelig adgang"*
- That is, only one thread is allowed to be executing in a critical region at a time...
- *Also called : Mutual exclusion*

The Lock

- Guard critical regions
- Example:
 - Our two threads will try to invoke `deposit()` and `withdraw()`
- But our account object, `a`, has an "lock" associated
 - Only one thread may acquire the lock at any time!

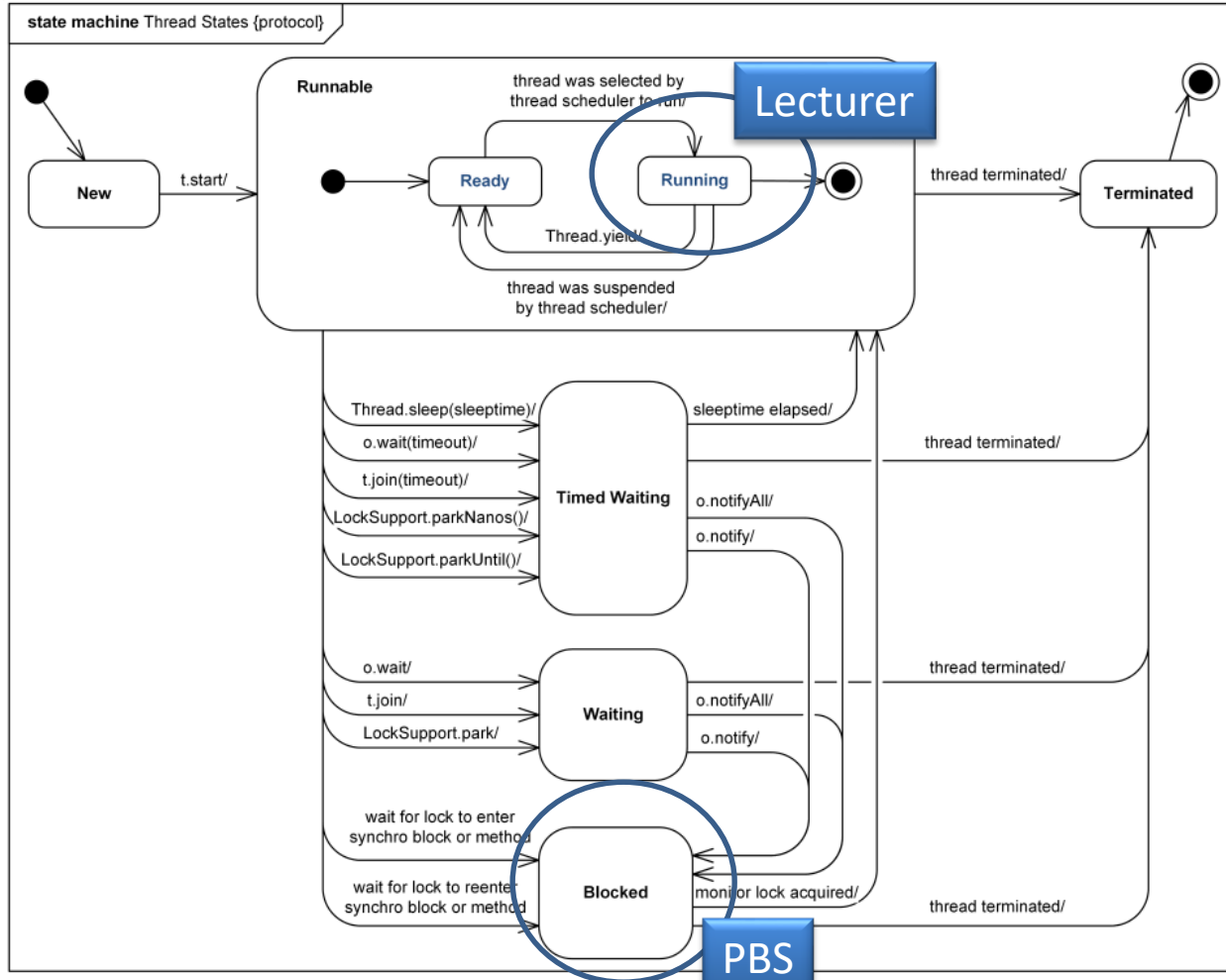


In 'deposit()'

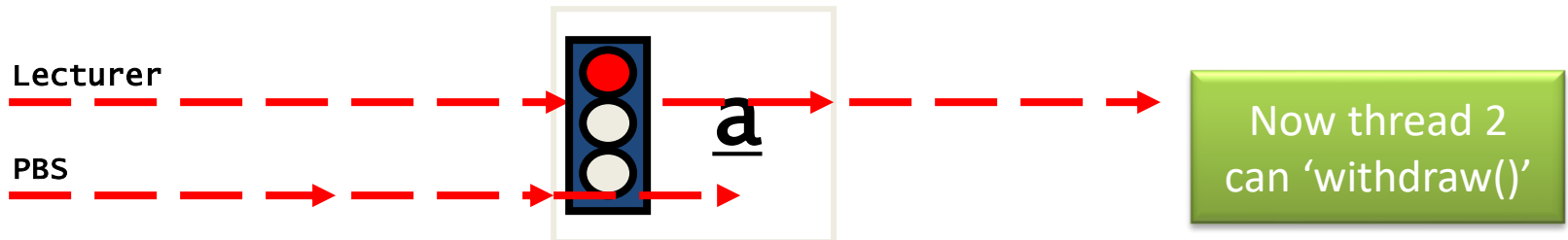
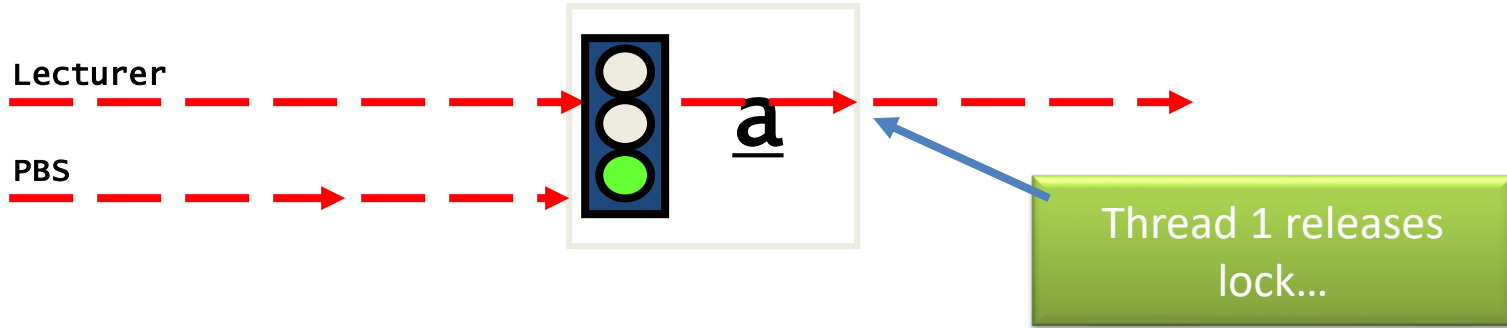




Thread State



And next...





Monitor/Synchronized

- A *Monitor* is a class whose methods are *all* associated with a lock/semaphore/"lysregulering"
- In Java it is more fine-grained: **synchronized**
- Only methods with the **synchronized** keyword will respect the lock
- **There is only one lock per object !!!**
 - Not one per method!!!

Synchronized

- Synchronized methods = whole method is a critical sect.

```
public synchronized void add(int value){  
    this.count += value;  
}
```

- *Do not make too much code a critical section!*
 - It slows a program down if all threads have to wait for the lock
- Rule:
 - Only **writing** needs to be guarded
- Rule 2:
 - Oh yeah – and **reads** of items more than N bits
 - N = 16, 32, 64 - Depending upon your processor !!!

Java is portable
code?



Synchronized Section

- You can simply state the object you want to sync on, on a smaller portion of the code!

```
public void add(int value){  
  
    synchronized(this){  
        this.count += value;  
    }  
}
```




Example again:

AARHUS UNIVERSITET

- Exercise: What should be **synchronized**?

```
class Counter {
    private Integer count = new Integer(0);

    public void incrementCount() {
        count++;
    }
    public int getCount() {
        return count;
    }
}
```

```
d:\proj\frsproject\concurrency>javac CounterTest.java
d:\proj\frsproject\concurrency>java CounterTest
Count    : 10000000
Expected: 10000000

d:\proj\frsproject\concurrency>java CounterTest
Count    : 10000000
Expected: 10000000

d:\proj\frsproject\concurrency>java CounterTest
Count    : 10000000
Expected: 10000000
```



Server Side and Client Side

- So far, the object itself has stated its synchronization.
- Often, this is not ok.
- Consider this ATM code:
 - *Account.debit()* is **synchronized**
- *What is the problem???*

```
long available = account.getBalance();
if(available > 0){
    System.out.print("You have "+available+
        " available, how much do you want? ");
    long amount = keyboard.nextLong();
    if(amount <= 0){
        System.out.println("The amount must be positive.");
    }
    else if(amount > available){
        System.out.println("That is too much.");
    }
    else{
        // This should be ok, but ...
        if(!account.debit(amount)){
            System.out.println("The ATM debit failed!");
        }
    }
}
```



Server Side and Client Side

- *What is the problem???*

```
long available = account.getBalance();
if(available > 0){
    System.out.print("You have "+available+
        " available, how much do you want? ");
    long amount = keyboard.nextLong();
    if(amount <= 0){
        System.out.println("The amount must be positive.");
    }
    else if(amount > available){
        System.out.println("That is too much.");
    }
    else{
        // This should be ok, but ...
        if(!account.debit(amount)){
            System.out.println("The ATM debit failed!");
        }
    }
}
```



Client Side Synchronization

- Synchronized takes the object as parameter, thus
- `synchronized(account) {`
- `<<<ATM code here>>>`
- `}`
- Will solve it – it is executed atomically using the lock of the account...

- Hey – did we not miss a thing here???

- Thread starts ‘deposit()’
 - Acquire the lock
- Then calls ‘setBalance()’
 - But the lock is taken???

```
    synch
public boolean deposit(long amount){
    if(amount >= 0){
        return setBalance(getBalance()+amount);
    }
    else {
        return false;
    }
}
```

- **Reentrant critical sections**

- If a thread ‘t’ has acquired the lock on object ‘a’, then it is free to invoke all other synchronized methods in ‘a’
 - Reentrance: The thread may *reenter* in locked methods
 - Recursion !



AARHUS UNIVERSITET

And – moving on...

- Java 5 onwards introduced 'java.util.concurrent' because the old 'synchronized' was way too simple...
 - A lot of concurrent data structures
 - An fine-grained 'synchronized' **Lock**

Main Differences Between Locks and Synchronized Blocks

The main differences between a `Lock` and a synchronized block are:

- A synchronized block makes no guarantees about the sequence in which threads waiting to enter it are granted access.
- You cannot pass any parameters to the entry of a synchronized block. Thus, having a timeout trying to get access to a synchronized block is not possible.
- The synchronized block must be fully contained within a single method. A `Lock` can have its calls to `lock()` and `unlock()` in separate methods.

ReentrantLock

- Instead of 'synchronized' on getCount()

- Note:

- Explicit lock object!
- Not on 'this'...

- Can have more locks in play in same object!

```
private final ReentrantLock lock = new ReentrantLock();
private int count = 0;

//Locking using Lock and ReentrantLock
public int getCount() {
    lock.lock();
    try {
        System.out.println(Thread.currentThread().getName() + " gets Count: " + count);
        return count++;
    } finally {
        lock.unlock();
    }
}
```

Ups. Uncle Bob would not be happy about this code!

- Or
 - tryLock(100 ms)
 - Will time out waiting to get the lock
 - Avoid indefinite waiting for a lock that a frozen thread has taken!
- Liability
 - *Code readability suffers greatly!*
 - *Some day you will forget the finally clause ☹️*

```
lock.lock();
try {
    // critical region
} finally {
    lock.unlock();
}
```

Use the 'try-with-resources'!



AARHUS UNIVERSITET

Deadlocks

Availability AntiPattern:
Blocked Threads

- **Deadlock**

- *A thread waits infinitely for an event that will never happen*

- That is

- T1, is in synchronized method in A, call sync. method in B
 - T2, has acquired lock on B, and waits for lock on C
 - ...
 - T4 waits for lock on A – which T1 has !

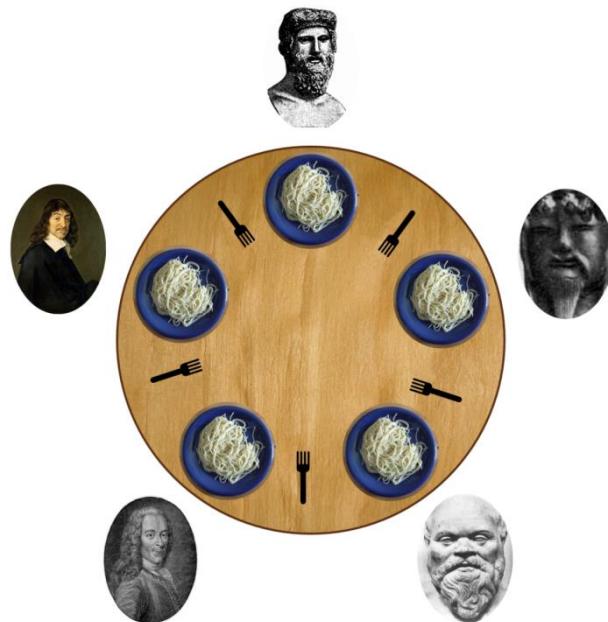
- Result

- Utterly nothing!!!

```
Thread 1  locks A,  waits for B
Thread 2  locks B,  waits for C
Thread 3  locks C,  waits for D
Thread 4  locks D,  waits for A
```

A Classic

- Edger Dijkstra, 1965: Dining Philosophers Problem
 - *Eat or think*
 - *To eat you need two forks*
- Design an algorithm that does not deadlock...



[By Benjamin D. Esham / Wikimedia Commons, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=56559>]

The General Rules of Thumb

- *Always acquire the locks in the same order*
 - Example: A, then B, then C

Note – does not work for the
Philosophers!

- Only works if you know the order ahead of time
- *Use timeouts on locks*
 - *If timeout: free all locks, wait, and retry...*

```
Thread 1:  
  
lock A  
lock B  
  
Thread 2:  
  
wait for A  
lock C (when A locked)  
  
Thread 3:  
  
wait for A  
wait for B  
wait for C
```