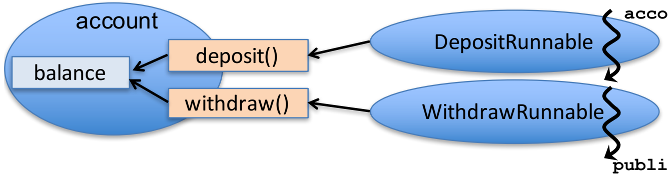
Deadlock

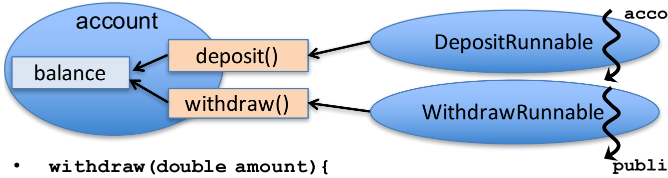
DeadlockedBankAccount.java



```
public void run(){
 for(int i = 0; i < 10; i++){
 account.deposit(100); }
```

```
public void run(){
 for(int i = 0; i < 10; i++){
 account.withdraw(100); }
```

DeadlockedBankAccount.java



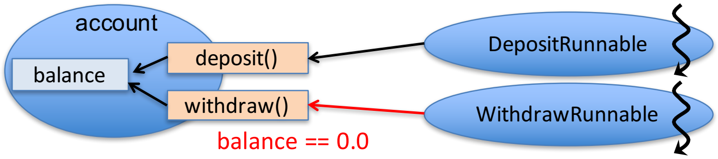
```
public void run(){
 for(int i = 0; i < 10; i++){
 account.deposit(100); }
```

```
public void run(){
 for(int i = 0; i < 10; i++){
 account.withdraw(100); }
```

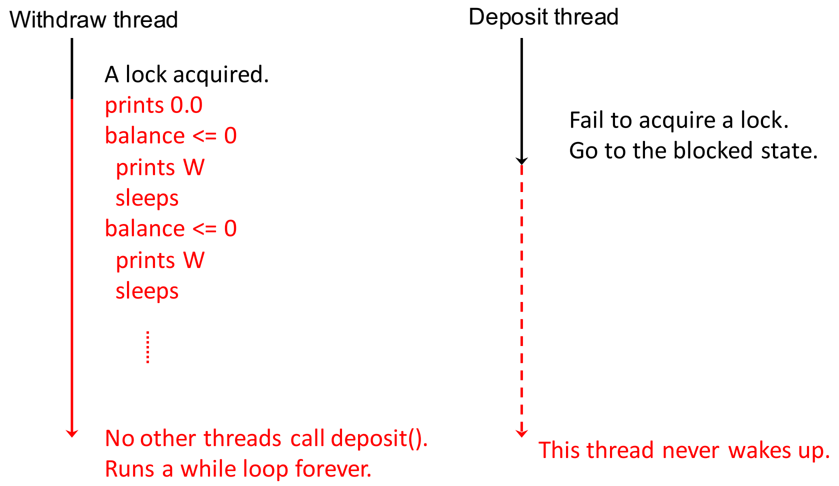
```
•   withdraw(double amount){
     lock.lock();
     while(balance <= 0){
         System.out.print("W");
         // waiting for the balance to exceed 0
         Thread.sleep(1000);
     }
     balance -= amount;
     lock.unlock(); }

•   deposit(double amount){
     lock.lock();
     while(balance > 10000){
         System.out.print("W");
         // waiting for the balance to go below 10,000
         Thread.sleep(1000)
     }
     balance += amount;
     lock.unlock(); }
```

Deadlock

- Assume the withdrawal thread goes ahead and runs first.
- Output
  - Lock obtained
    Current balance (w): 0.0WWWWWWWW



- Current balance (w): 0.0WWWWWWWW



DeadlockedBankAccount2.java

- Previous version

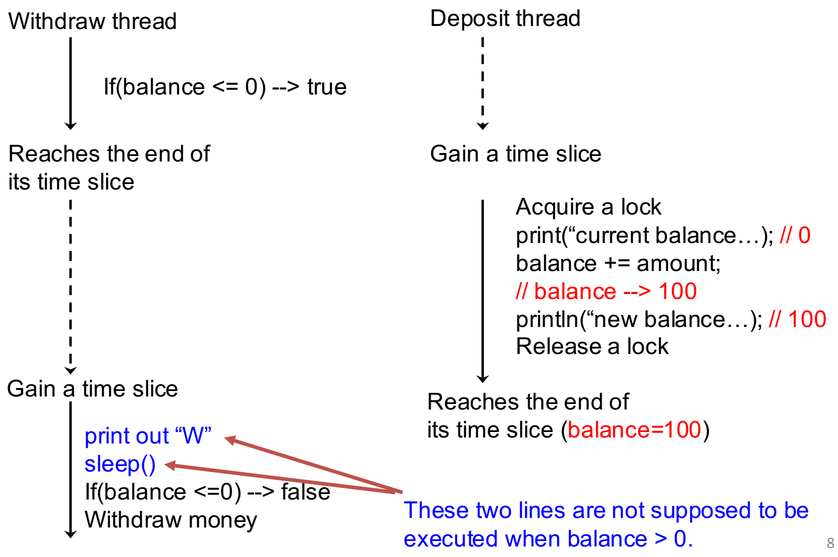
```
– withdraw(double amount){
    lock.lock();
    while(balance <= 0){
        System.out.print("W");
        // waiting for the
        // balance to exceed 0
        Thread.sleep(1000);
    }
    balance -= amount;
    lock.unlock();
  }

– deposit(double amount){
    lock.lock();
    while(balance > 10000){
        System.out.print("W");
        // waiting for the balance
  to go below 10,000
        Thread.sleep(1000)
    }
    balance += amount;
    lock.unlock(); }
```

- New version

```
– withdraw(double amount){
    while( balance <= 0 ){
        System.out.print("W");
        Thread.sleep(2);
    }
    lock.lock();
    balance -= amount;
    lock.unlock();
  }

– deposit(double amount){
    while( balance > 10000 ){
        System.out.print("W");
        Thread.sleep(2);
    }
    lock.lock();
    balance += amount;
    lock.unlock();
  }
```

- Has no deadlock problems.
- Can generate race conditions.

A Potential Race Condition in DeadlockedBankAccount2

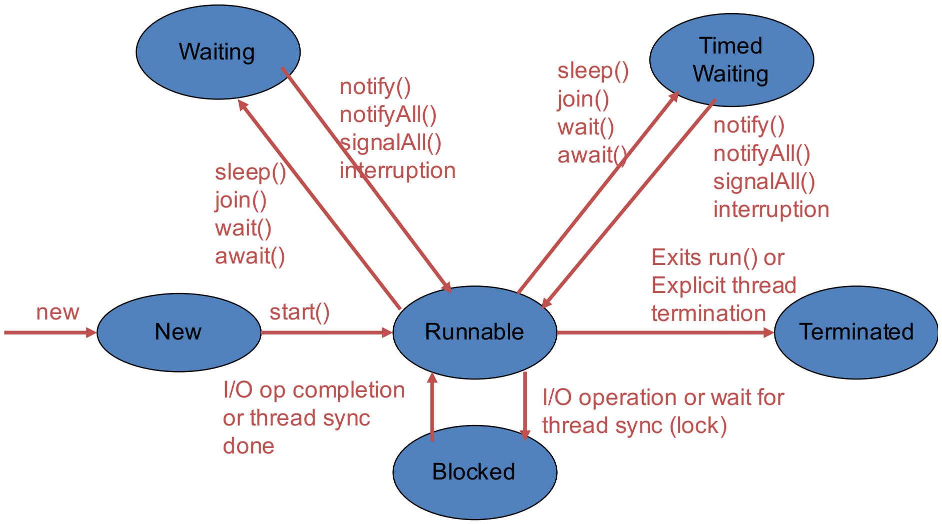

Withdraw thread
If(balance <= 0) --> true
Reaches the end of its time slice
Gain a time slice
print out "W"
sleep()
If(balance <=0) --> false
Withdraw money
Deposit thread
Gain a time slice
Acquire a lock
print("current balance…); // 0
balance += amount;
// balance --> 100
println("new balance…); // 100
Release a lock
Reaches the end of its time slice (balance=100)
These two lines are not supposed to be executed when balance > 0.

Avoiding Deadlocks and Race Conditions

- Use a Condition object(s).
  - Allow a thread to
    - temporarily release a lock so that another thread can proceed
      - The thread goes to the Waiting state from the Runnable state.
    - re-acquire the lock later.

- java.util.concurrent.locks.Condition
  - Obtain its instance from a lock object

```
ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition(); //factory method
condition.await();
```



ThreadSafeBankAccount2

-
  ```
  Condition sufficientFundsCondition = lock.newCondition();
  Condition belowUpperLimitFundsCondition = lock.newCondition();
  ```

-
  ```
  withdraw(double amount){
   lock.lock();
   while(balance <= 0){
       // waiting for the balance to exceed 0
      sufficientFundsCondition.await(); }
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock(); }
  ```

-
  ```
  deposit(double amount){
   lock.lock();
   while(balance >= 300){
      // waiting for the balance to go below 10000.
      belowUpperLimitFundsCondition.await(); }
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock(); }
  ```

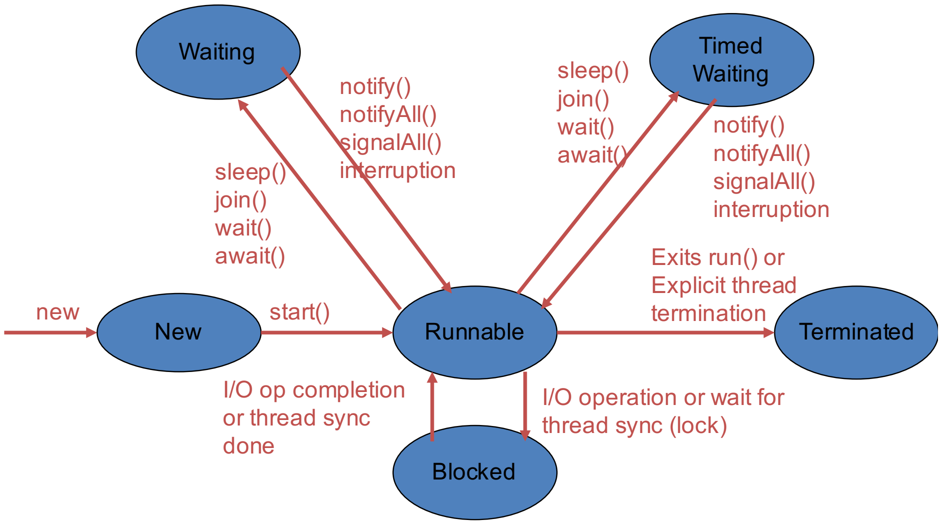
ThreadSafeBankAccount2

- Output

```
  – Lock obtained
  – 7 (d): current balance: 0.0
  – 7 (d): new balance: 100.0
  – Lock released
  – Lock obtained
  – 8 (d): current balance: 100.0
  – 8 (d): new balance: 200.0
  – Lock released
  – Lock obtained
  – 9 (d): current balance: 200.0
  – 9 (d): new balance: 300.0
  – Lock released
  – Lock obtained
  – 10 (d): current balance: 300.0
  – 10 (d): await(): Balance has reached the upper limit.
  – Lock obtained
  – 11 (d): current balance: 300.0
  – 11 (d): await(): Balance has reached the upper limit.
  – Lock obtained
  – 12 (w): current balance: 300.0
  – 12 (w): new balance: 200.0
  – Lock released
  – 10 (d): new balance: 300.0
  – Lock released
  – 11 (d): await(): Balance has reached the upper limit.
```

Condition

- await()
  - Will be waiting until it is signaled or interrupted
  - Will be waiting until it is signaled or interrupted, or until a specified waiting time (relative time) elapsed.
  - Will be waiting until it is signaled or interrupted, or until a specified deadline (absolute time).
  - If signaled, goes to the Runnable state and re-acquires a lock.
    - Will be "blocked" if fails to re-acquire the lock.
  - Throws an InterruptedException if interrupted.
    - c.f. previous lecture note that explains InterruptedException
- signalAll()
  - Wakes up all waiting threads on a condition object.
    - All of them go to the "runnable" state.
    - One of them will re-acquire a lock.



- When a thread calls await(), signal() or signalAll() on a Condition object,
  - the thread is assumed to hold the lock associated with the Condition object.
- If the thread does not,
  - an IllegalMonitorStateException is thrown.

SignalAll() Before or After a State Change?

```
• withdraw(double amount){
   lock.lock();
   while(balance =< 0){
        // waiting for the balance to exceed 0
       sufficientFundsCondition.await();  }
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock();  }
• deposit(double amount){
   lock.lock();
   while(balance >= 300){
       // waiting for the balance to go below 300.
       belowUpperLimitFundsCondition.await();  }
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock();  }
```

- What happens if you call signalAll() first and then update the balance? Any problems?

-
  ```
  withdraw(double amount){
   lock.lock();
   while(balance =< 0){
       // waiting for the balance to exceed 0
      sufficientFundsCondition.await(); }
   belowUpperLimitFundsCondition.signalAll();
   balance -= amount;
   lock.unlock(); }
  ```

-
  ```
  deposit(double amount){
   lock.lock();
   while(balance >= 300){
      // waiting for the balance to go below 300.
      belowUpperLimitFundsCondition.await(); }
   sufficientFundsCondition.signalAll();
   balance += amount;
   lock.unlock(); }
  ```
- Need to worry about race conditions in this case?

(1) W thread: “waiting” temporarily releases the lock

```
withdraw(double amount){
 lock.lock();
 while(balance =< 0){
     // waiting for the balance to exceed 0
    sufficientFundsCondition.await(); }
 belowUpperLimitFundsCondition.signalAll();
 balance -= amount;
 lock.unlock(); }
```

(2) D thread: signalAll(). Ctx switch

```
deposit(double amount){
 lock.lock();
 while(balance >= 300){
     // waiting for the balance to go below 300.
    belowUpperLimitFundsCondition.await(); }
sufficientFundsCondition.signalAll();
 balance += amount;
 lock.unlock(); }
```

- A context switch can occur in between signalAll() and state change?
- A “W” thread can withdraw money before a “D” thread deposits money?
  - Can the balance variable have a negative value?

(1) W thread: “waiting” temporarily releases the lock

```
withdraw(double amount){
 lock.lock();
 while(balance =< 0){
     // waiting for the balance to exceed 0
    sufficientFundsCondition.await(); }
 belowUpperLimitFundsCondition.signalAll();
 balance -= amount;
 lock.unlock(); }
```

(3) W thread: “runnable” Tries to acquire the lock again and fails. Goes to “blocked.”

(2) D thread: signalAll(). Ctx switch

```
deposit(double amount){
 lock.lock();
 while(balance >= 300){
     // waiting for the balance to go below 300.
    belowUpperLimitFundsCondition.await(); }
sufficientFundsCondition.signalAll();
 balance += amount;
 lock.unlock(); }
```

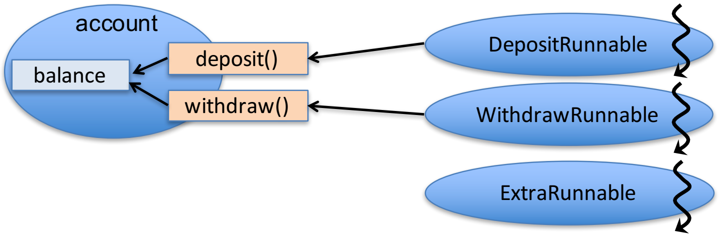
- A “W” thread CANNOT withdraw money before a “D” thread deposits money.
- A “D” thread CANNOT deposit money before a “W” thread withdraws money.

Two Important Things (1)

- A state change (or value change) can be made on a shared variable safely after calling signalAll()
  - AS FAR AS the state changes in atomic code
- Common programming convention/practice:
  - A state change first, followed by signalAll().

Two Important Things (2)

- A JVM DOES context switches even when a thread runs in atomic code.
  - A lock guarantees that only one thread exclusively runs atomic code at a time.



Note that…

- Some books and online materials explicitly/implicitly say that context switches never occur when a thread runs in atomic code.
- It is wrong!

signal() and signalAll()

- signalAll()
  - Wakes up all waiting threads on a condition object.
    - All of them go to the "runnable" state.
    - One of them will re-acquire a lock.
- signal()
  - Wakes up one of waiting threads on a condition object.
    - One of them goes to the "runnable" state. The others stay at the "waiting" state.
    - JVM's thread scheduler selects one of them. Assume a random selection.
      - Not predictable which waiting thread to be selected.

ThreadSafeBankAccount2

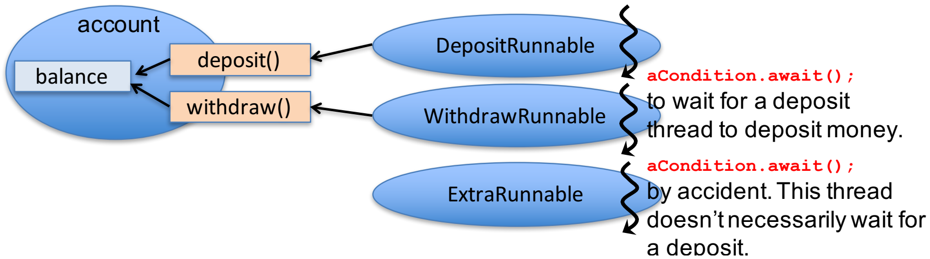
-
  ```
  Condition sufficientFundsCondition = lock.newCondition();
  Condition belowUpperLimitFundsCondition = lock.newCondition();
  ```

-
  ```
  withdraw(double amount){
   lock.lock();
   while(balance =< 0){
       // waiting for the balance to exceed 0
      sufficientFundsCondition.await(); }
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock(); }
  ```

-
  ```
  deposit(double amount){
   lock.lock();
   while(balance >= 300){
      // waiting for the balance to go below 300.
      belowUpperLimitFundsCondition.await(); }
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock(); }
  ```

signal() or signalAll()?

- signal() is more lightweight than signalAll().
  - Waking up a waiting thread is computationally expensive.
  - signal() wakes up only one thread.
- signalAll() is more *protective*.
  - signalAll() should be favored (at least in my personal taste).

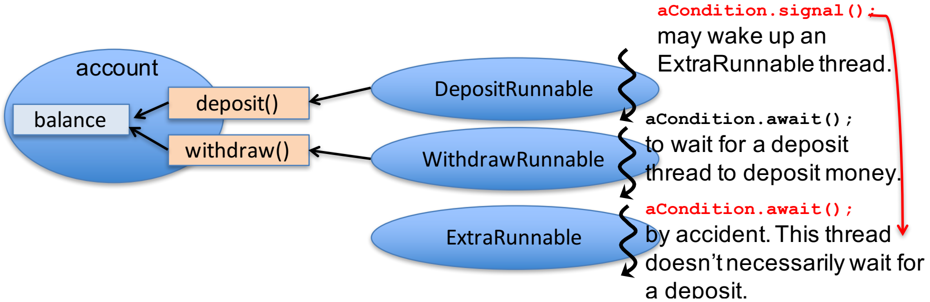


aCondition.await(); to wait for a deposit thread to deposit money.

aCondition.await(); by accident. This thread doesn't necessarily wait for a deposit.

signal()

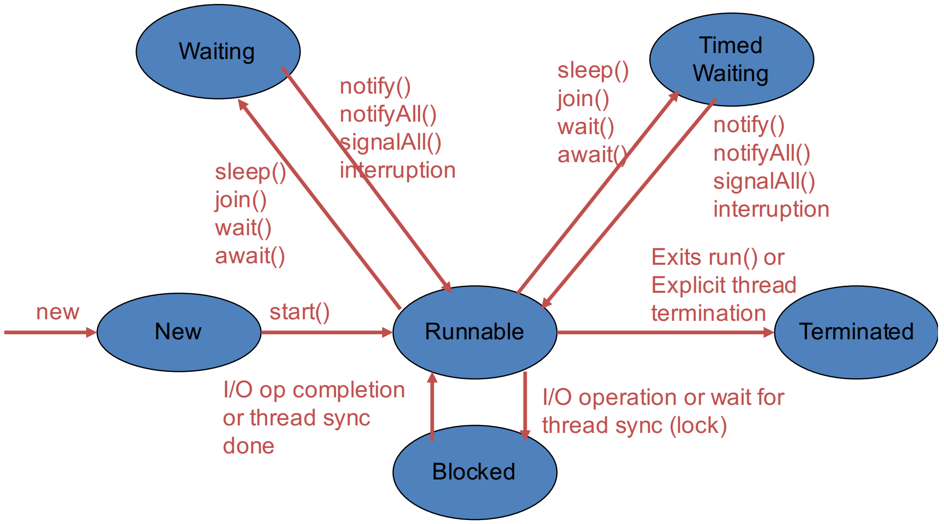
- A deposit thread may wake up an ExtraRunnable thread by calling signal().
  - A withdraw thread
    - loses a chance to withdraw money even if some money in the account.
    - may never be waked up if no threads call deposit afterward.



aCondition.signal(); may wake up an ExtraRunnable thread.

aCondition.await(); to wait for a deposit thread to deposit money.

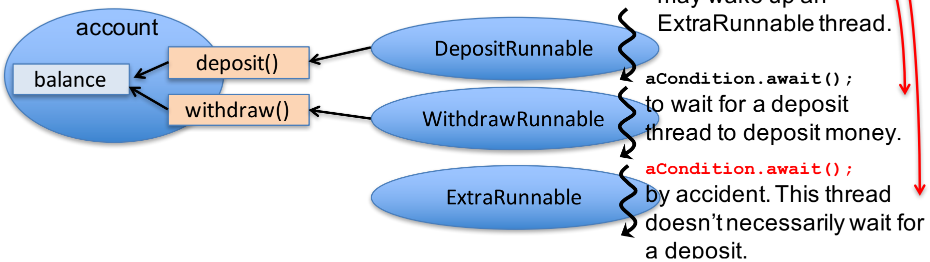
aCondition.await(); by accident. This thread doesn't necessarily wait for a deposit.



signalAll()

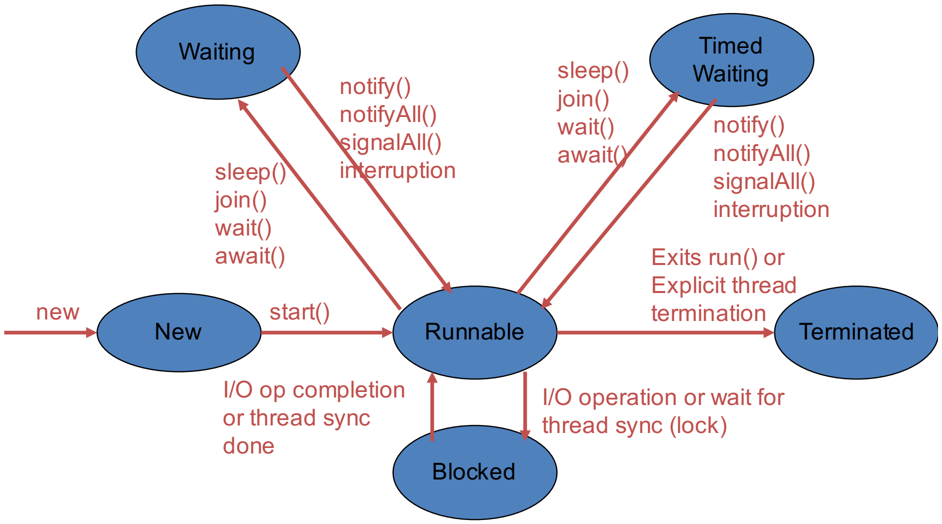
- A deposit thread wake up both a withdraw thread and an ExtraRunnalbe thread.
  - Both threads go into the "Runnable" state.
- If the ExtraRunnable thread acquires the lock associated with `aCondition`,
  - the withdraw thread
    - goes to the "blocked" state.
    - acquires the target lock when the ExtraRunnable releases it.
    - will wake up sooner or later.
    - never lose a chance to withdraw money.

aCondition.signalAll(); may wake up an ExtraRunnable thread.



aCondition.await(); to wait for a deposit thread to deposit money.

aCondition.await(); by accident. This thread doesn't necessarily wait for a deposit.



ThreadSafeBankAccount2

```
• Condition sufficientFundsCondition = lock.newCondition();
  Condition belowUpperLimitFundsCondition = lock.newCondition();

• withdraw(double amount){
   lock.lock();
   while(balance =< 0){
       // waiting for the balance to exceed 0
      sufficientFundsCondition.await();  }
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock();  }

• deposit(double amount){
   lock.lock();
   while(balance >= 300){
      // waiting for the balance to go below 300.
      belowUpperLimitFundsCondition.await();  }
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock();  }
```

“while” or “if” to Surround await()?

```
• withdraw(double amount){
   lock.lock();
   while(balance =< 0){
       // waiting for the balance to exceed 0
      sufficientFundsCondition.await();  }
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock();  }

• withdraw(double amount){
   lock.lock();
   if(balance =< 0){
       // waiting for the balance to exceed 0
      sufficientFundsCondition.await();  }
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock();  }
```

- “while” should be used rather than “if” when multiple threads call withdraw() concurrently. Why?

Problem

(1) b==0.
Two W
threads:
“waiting”

```
• withdraw(double amount){
   lock.lock();
   if(balance =< 0){
       // waiting for the balance to excee
      sufficientFundsCondition.await();  }
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll(
   lock.unlock();  }
```

(3) Two W threads:
“runnable” One of them
acquires the lock again
and releases it.
b==0.
The other W thread:
“blocked” on acquiring
the lock.

```
• deposit(double amount){
   lock.lock();
   if(balance >= 300){
      // waiting for the balance to go below 300.
      belowUpperLimitFundsCondition.await();  }
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock();  }
```

(2) D thread:
signalAll()
followed by
unlock()
b==100

(1) b==0. Two W threads: “waiting”

```
• withdraw(double amount){
   lock.lock();
   if(balance =< 0){
       // waiting for the balance to exceed 0
      sufficientFundsCondition.await(); }
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock(); }

• deposit(double amount){
   lock.lock();
   if(balance >= 300){
      // waiting for the balance to go below 300.
      belowUpperLimitFundsCondition.await(); }
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock(); }
```

(2) D thread: signalAll() followed by unlock() b==100

(3) Two W threads: “runnable” One of them acquires the lock again and releases it. b==0. The other W thread: “blocked” on acquiring the lock.

(4) The 2nd W thread acquires the lock and releases it. b==-100.

(1) b=0. Two W threads: “waiting”

```
• withdraw(double amount){
   lock.lock();
   if(balance =< 0){
      // waiting for the balance to excee
     sufficientFundsCondition.await(); }
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll
   lock.unlock(); }

• deposit(double amount){
   lock.lock();
   if(balance >= 300){
     // waiting for the balance to go below 300.
     belowUpperLimitFundsCondition.await(); }
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock(); }
```

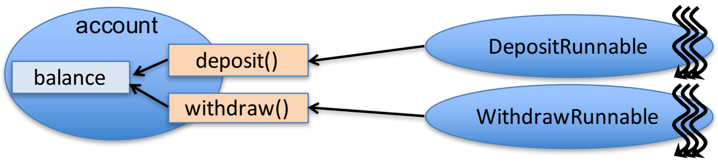
(2) D thread: signalAll() followed by unlock() balance==100

(3) Two W threads: “runnable” One of them acquires the lock again and releases it. balance==0. The other W thread: “blocked” on acquiring the lock.

(4) The 2nd W thread acquires the lock and releases it. b==-100.

- The 2nd thread should have made sure “balance>0.”
- If only one “W” thread runs, this problem does not occur.
- Always use a while loop.

ThreadSafeBankAccount2



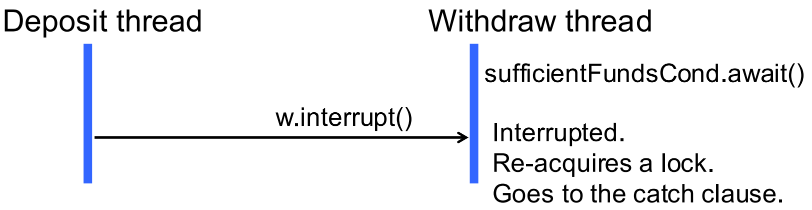
“if” or “while” in Atomic Code?

- You can use “if” rather than “while” for a conditional checking
  - if you use signal(), not signalAll().
- However, in practice, the while-signalAll pair is more common than the if-signal pair.

InterruptedException

- Some methods in Java API can throw InterruptedException.
  - Thread.sleep()
  - Thread.join()
  - Condition.await()
  - ReentrantLock.tryLock()

  - These methods can be long-running and cancellable.



Deposit thread

Withdraw thread

sufficientFundsCond.await()

w.interrupt()

Interrupted.
Re-acquires a lock.
Goes to the catch clause.

```
withdraw(double amount){
    lock.lock();
    while(balance =< 0){
        try{
            // waiting for the balance to exceed 0
            sufficientFundsCondition.await();
        }catch(InterruptedException e){
            //Do something
        }
    }
    ...}
```

Condition.await()

- await() lets the currently-executed thread to wait/sleep until another thread wakes it up with signal()/signalAll().

- interrupt() can interrupt a waiting/sleeping thread on a condition object.
  - The waiting/sleeping thread re-acquires a lock and throws an InterruptedException.
    - It does NOT immediately throw an InterruptedException.

```
withdraw(double amount){
    lock.lock();
    while(balance =< 0){
        try{
            // waiting for the balance to exceed 0
            sufficientFundsCondition.await();
        }catch(InterruptedException e){
            //Do something
        }
    }
    belowUpperLimitFundsCondition.signalAll();
    balance -= amount;
    lock.unlock(); }
```