Deadlock

CPSC 1181 – O.O.

Jeremy Hilliker

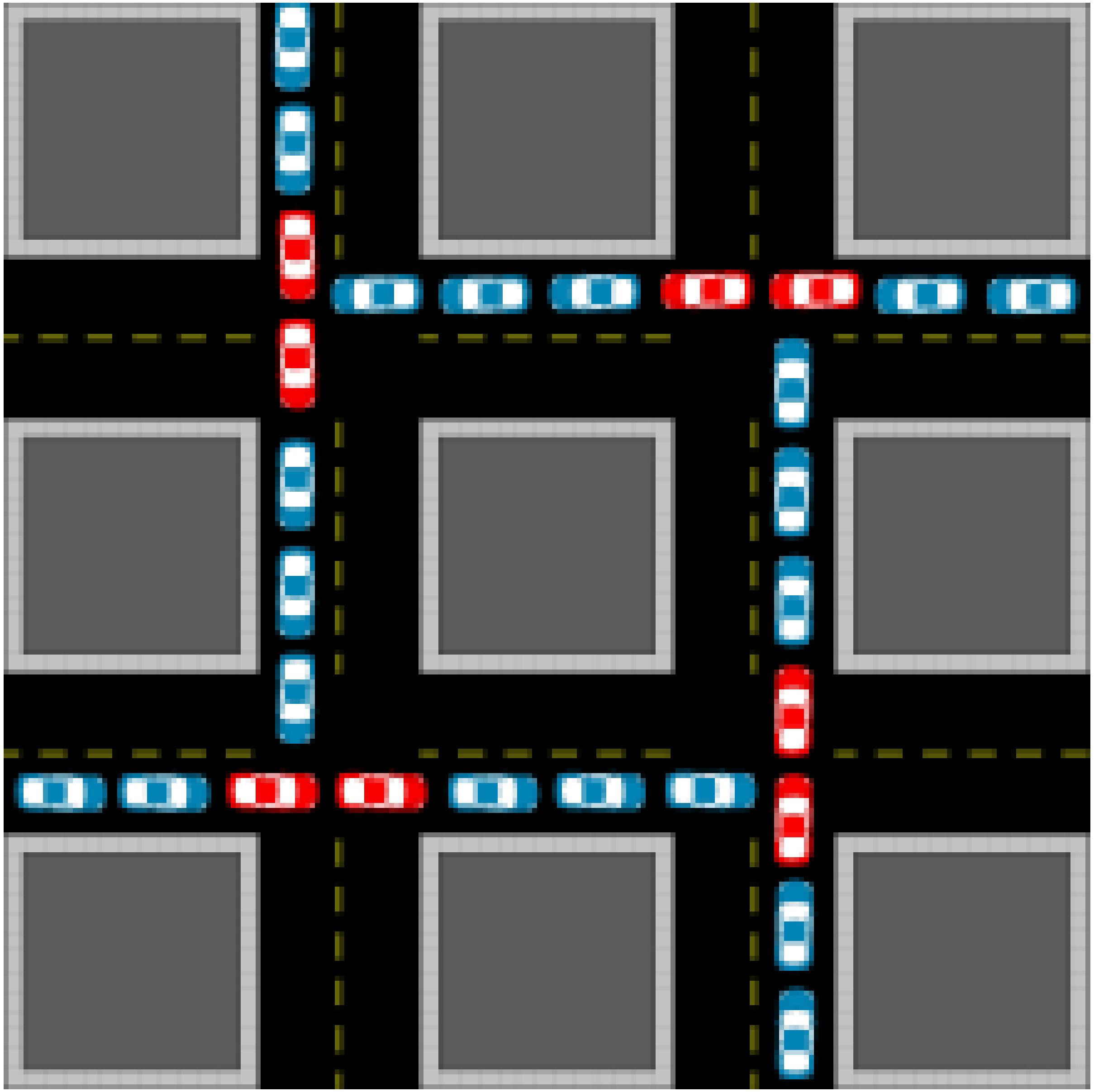
Summer 2017

Outline

- Deadlock
    - 1 lock
    - Condition variables
    - 2 locks
    - Livelock
    - Starvation
    - Priority Inversion

Deadlock

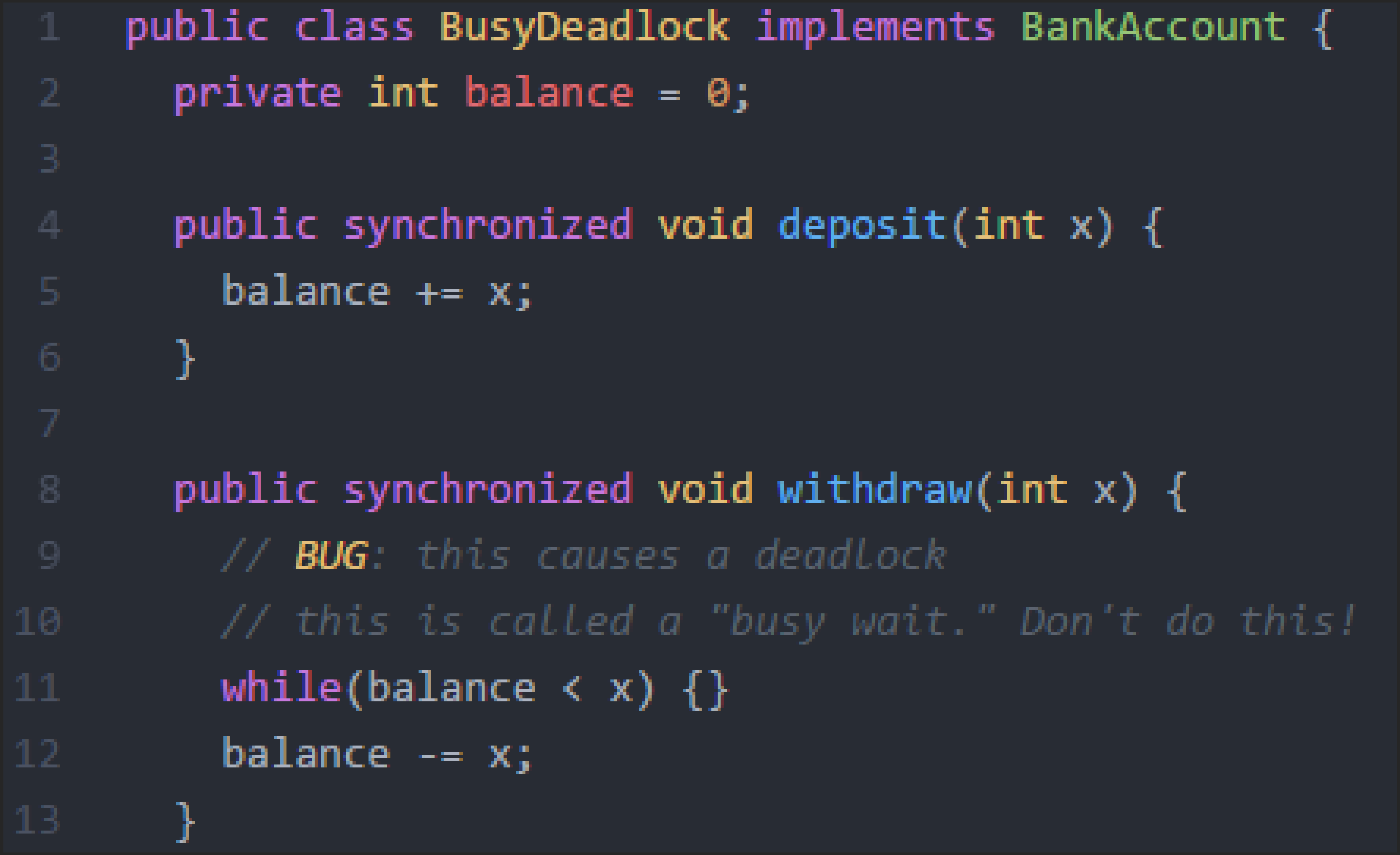
- <u>**Def'n**</u>: ***Deadlock*** occurs when two or more threads cannot proceed because they are all waiting for each other to release a lock.
  - These threads move to the WAIT state while they await a signal indicating that the lock has been released.



Causes

- More insidious: Contention over 2 locks
  - T1 acquires lock A, pre-empted
  - T2 acquires lock B, wants lock A, must WAIT
  - T1 resumes, wants lock B, must WAIT
  - T1&T2 stuck in WAIT state
- Less insidious: Contention over 1 lock
  - T1 acquires a lock
  - Notices that it cannot proceed until T2 is done
  - Waits for T2 to *while still holding the lock*
  - T2 attempts to do its work,
    - cannot because it needs the lock held by T1
  - T2 stuck in WAIT, T1 being dumb

Contention Over 1 Lock



```java
public class BusyDeadlock implements BankAccount {
  private int balance = 0;

  public synchronized void deposit(int x) {
    balance += x;
  }

  public synchronized void withdraw(int x) {
    // BUG: this causes a deadlock
    // this is called a "busy wait." Don't do this!
    while(balance < x) {}
    balance -= x;
  }
```

Condition Objects

- A thread temporarily releases a lock while it WAITs for a ***condition*** to be true
- Regains the lock later

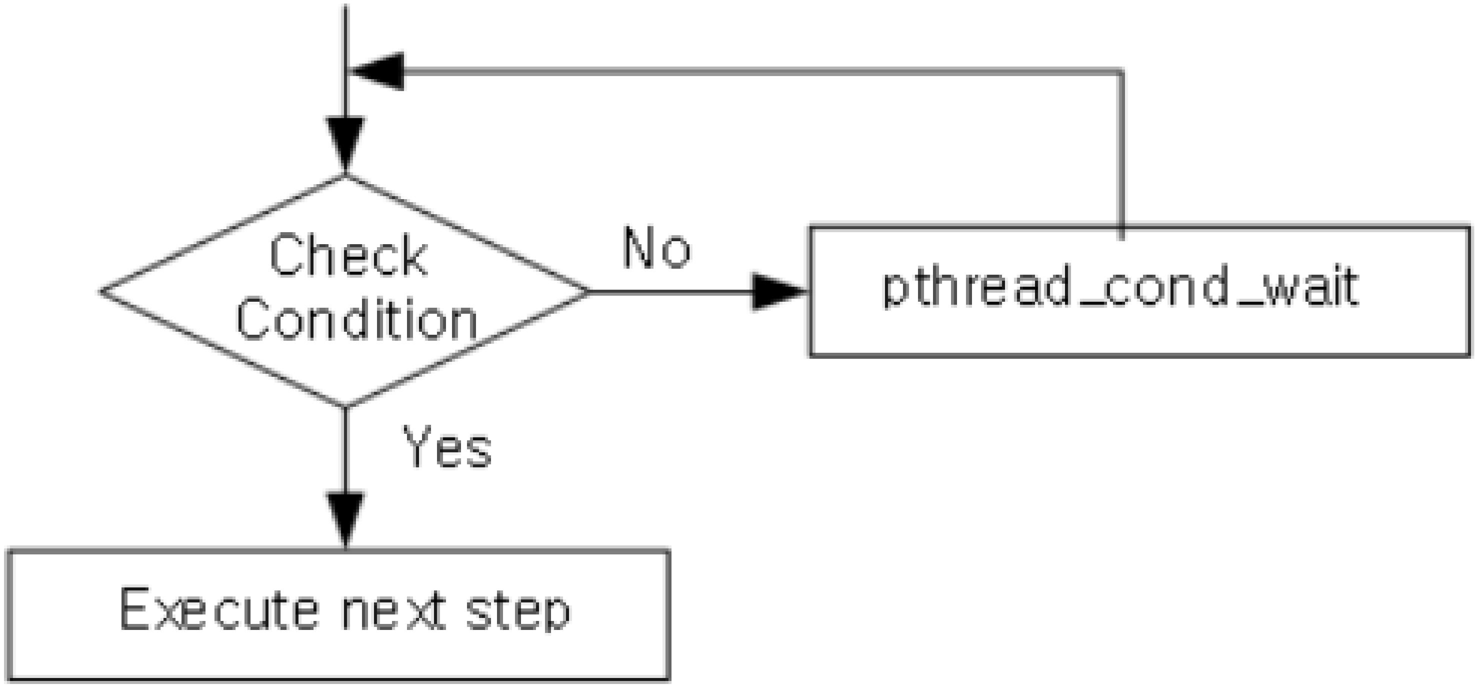
- A Condition belongs to one Lock
- A Lock may have 0 or more Conditions

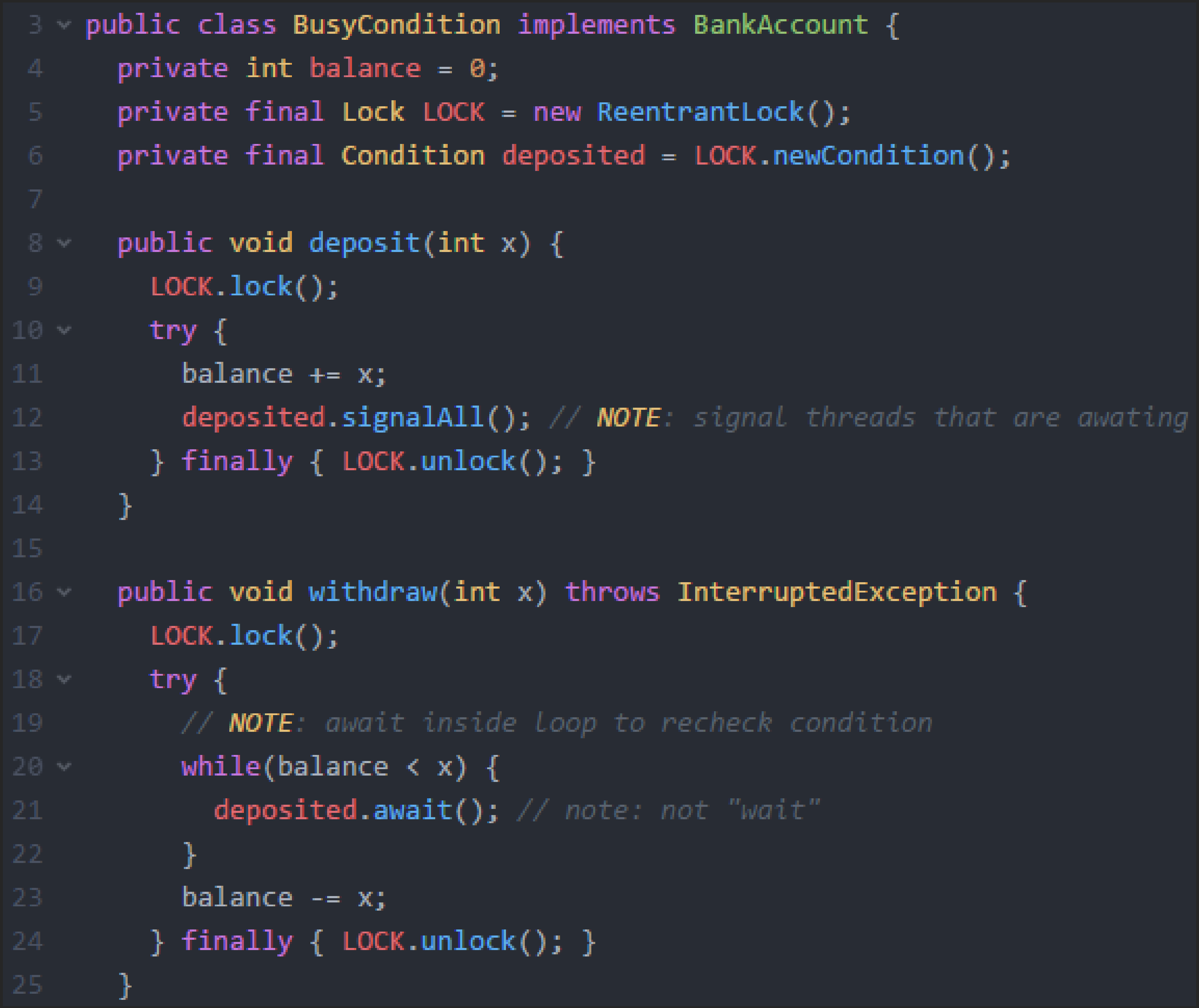
Condition Pattern:

- Make a lock
  - Create condition(s)

- Acquire lock
  - Realize you must wait
  - Call “await()” method on condition object
    - Note: not “wait()”!
  - Put that call inside a loop that retests the condition
    - For spurious wakes, and for signalAll

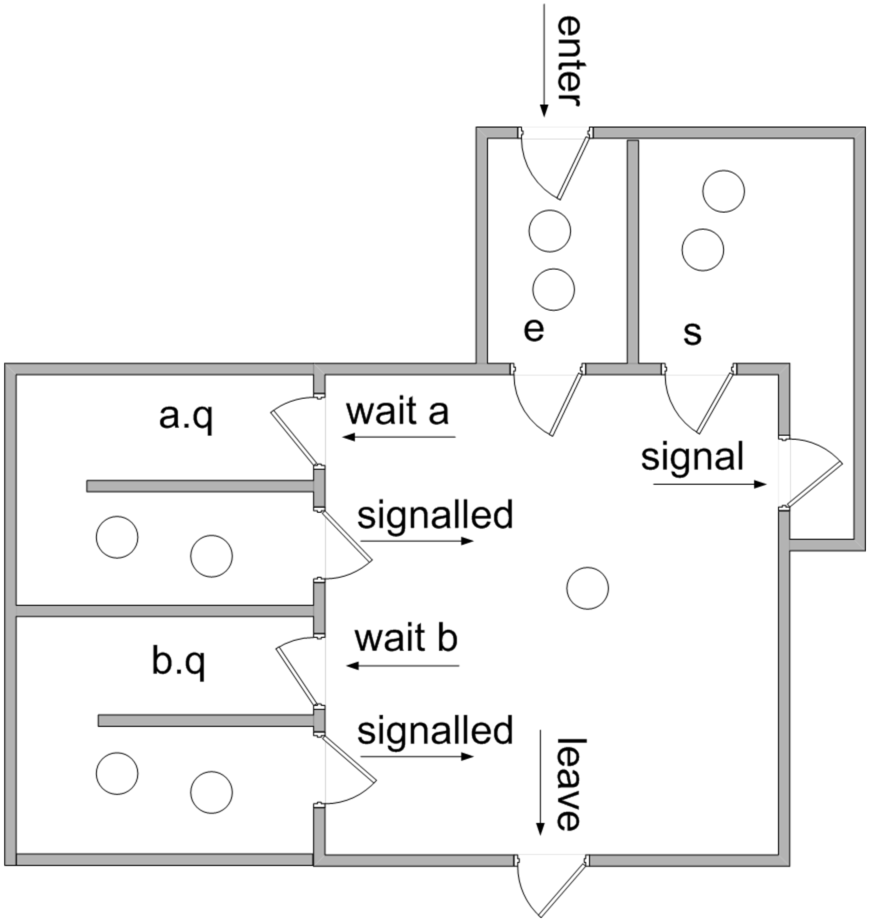
- Acquire lock
  - Do work
  - Signal *all* waiting threads
  - Release lock


Check Condition
No
pthread_cond_wait
Yes
Execute next step



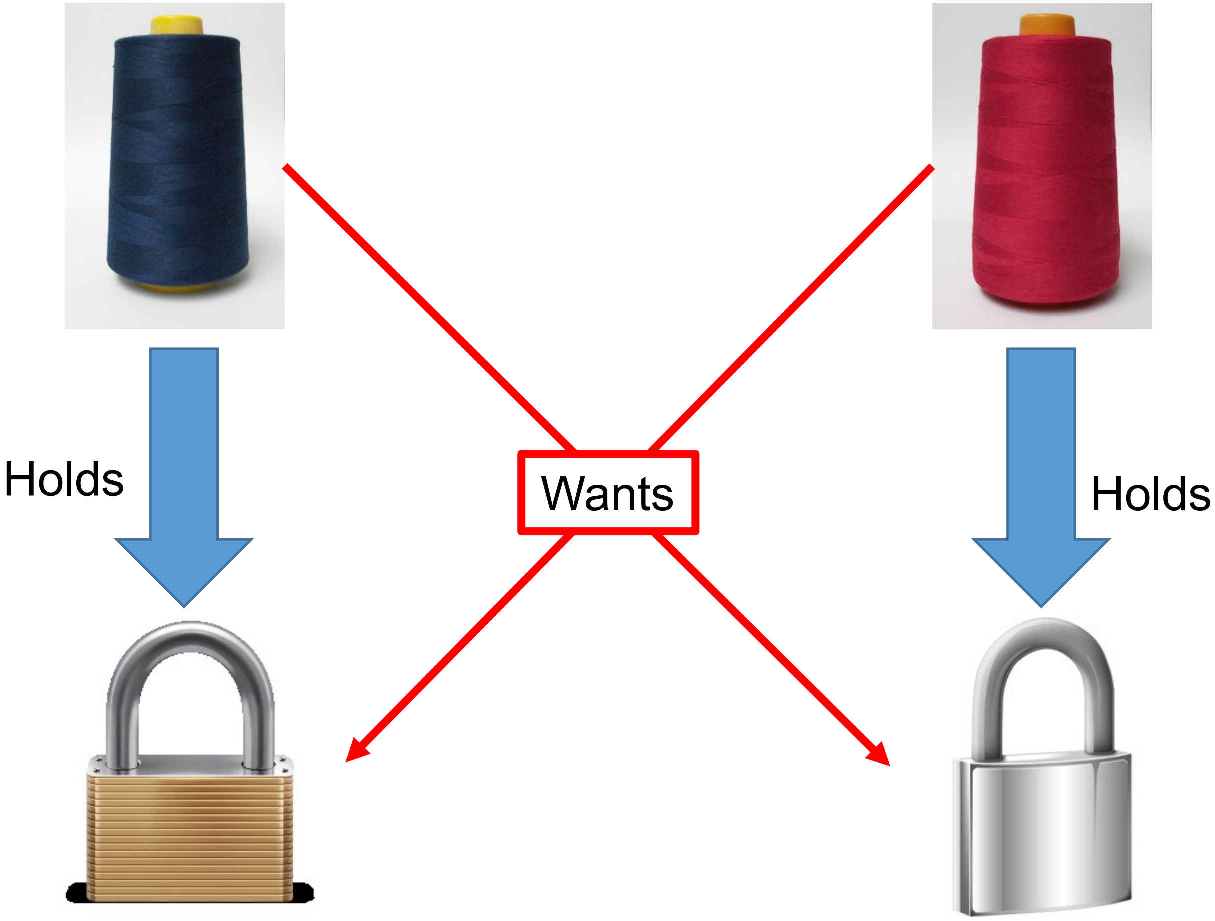
```java
public class BusyCondition implements BankAccount {
  private int balance = 0;
  private final Lock LOCK = new ReentrantLock();
  private final Condition deposited = LOCK.newCondition();

  public void deposit(int x) {
    LOCK.lock();
    try {
      balance += x;
      deposited.signalAll(); // NOTE: signal threads that are awating
    } finally { LOCK.unlock(); }
  }

  public void withdraw(int x) throws InterruptedException {
    LOCK.lock();
    try {
      // NOTE: await inside loop to recheck condition
      while(balance < x) {
        deposited.await(); // note: not "wait"
      }
      balance -= x;
    } finally { LOCK.unlock(); }
  }
```

Locks with Conditions as Monitors



A Hoare monitor
Diagram from Buhr

Contention Over 2 Locks



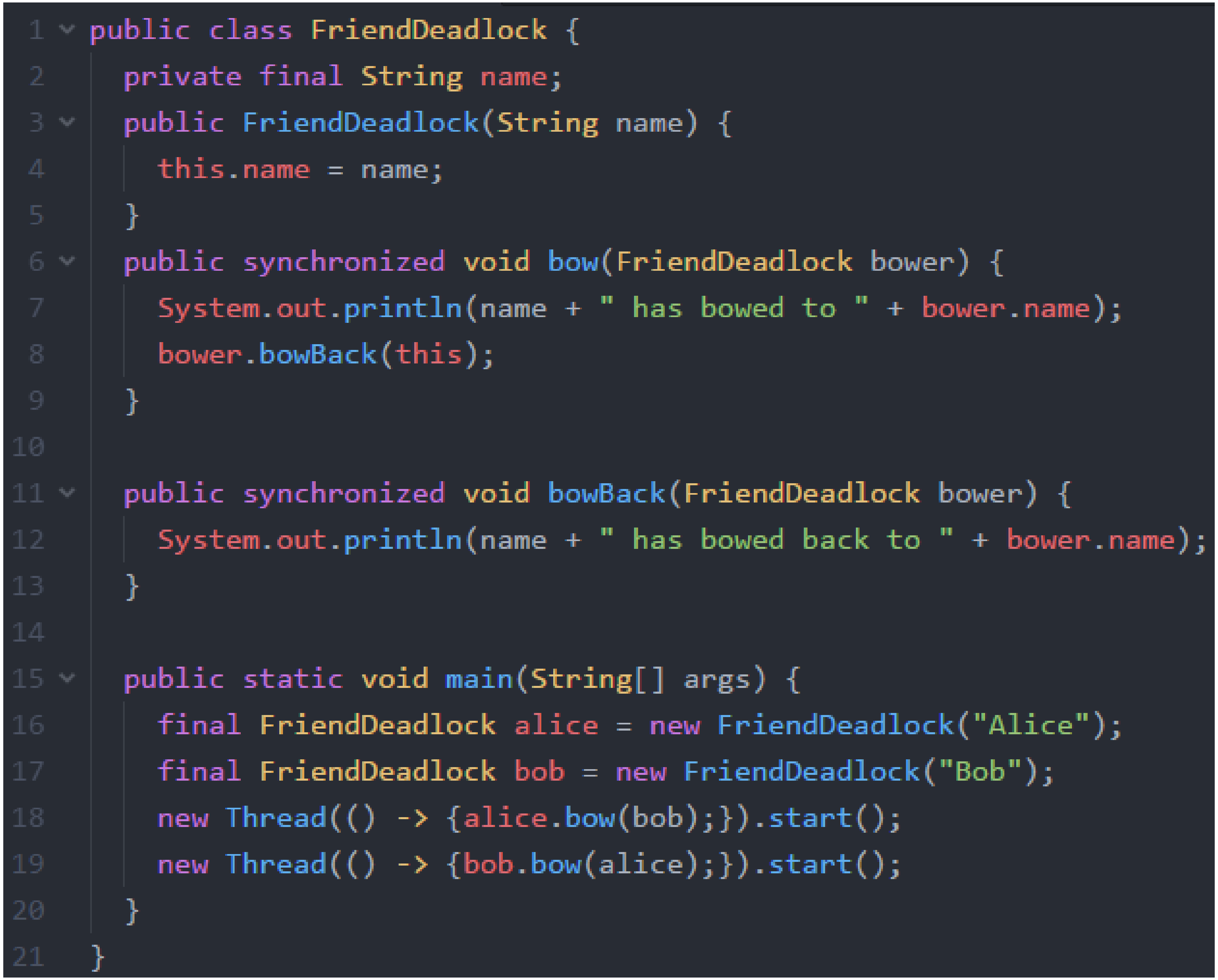
```
public static Object fooLock= new Object();
public static Object barLock= new Object();
// ...
public void foo() {
  synchronized (fooLock) {
    synchronized (barLock) {
      doSomething();
    }
  }
}
public void bar() {
  synchronized (barLock) {
    synchronized (fooLock) {
      doSomethingElse();
    }
  }
}
```

Thread A acquires fooLock
Thread B acquired barLock
Thread A wants barLock
Thread B wants fooLock

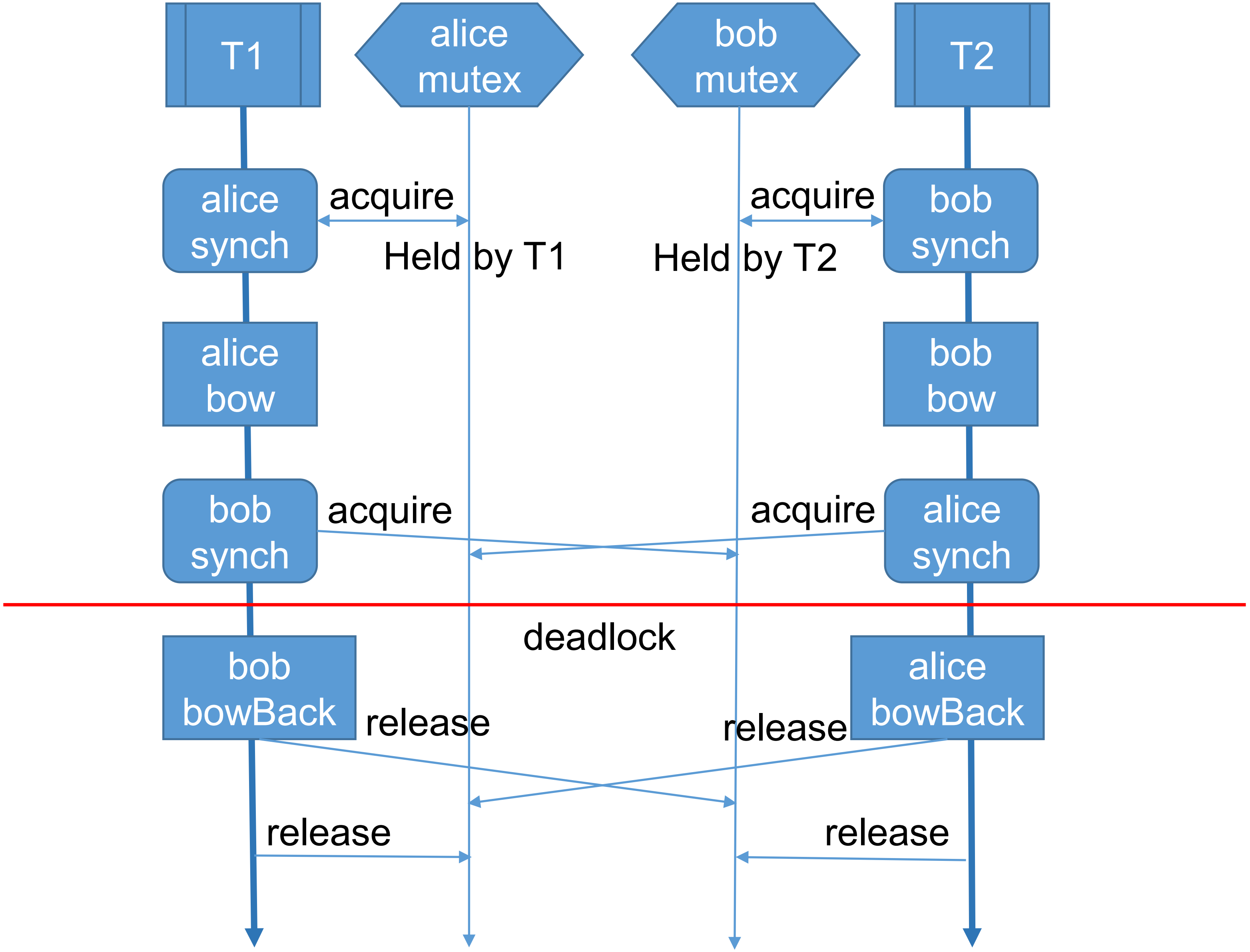
Problem:

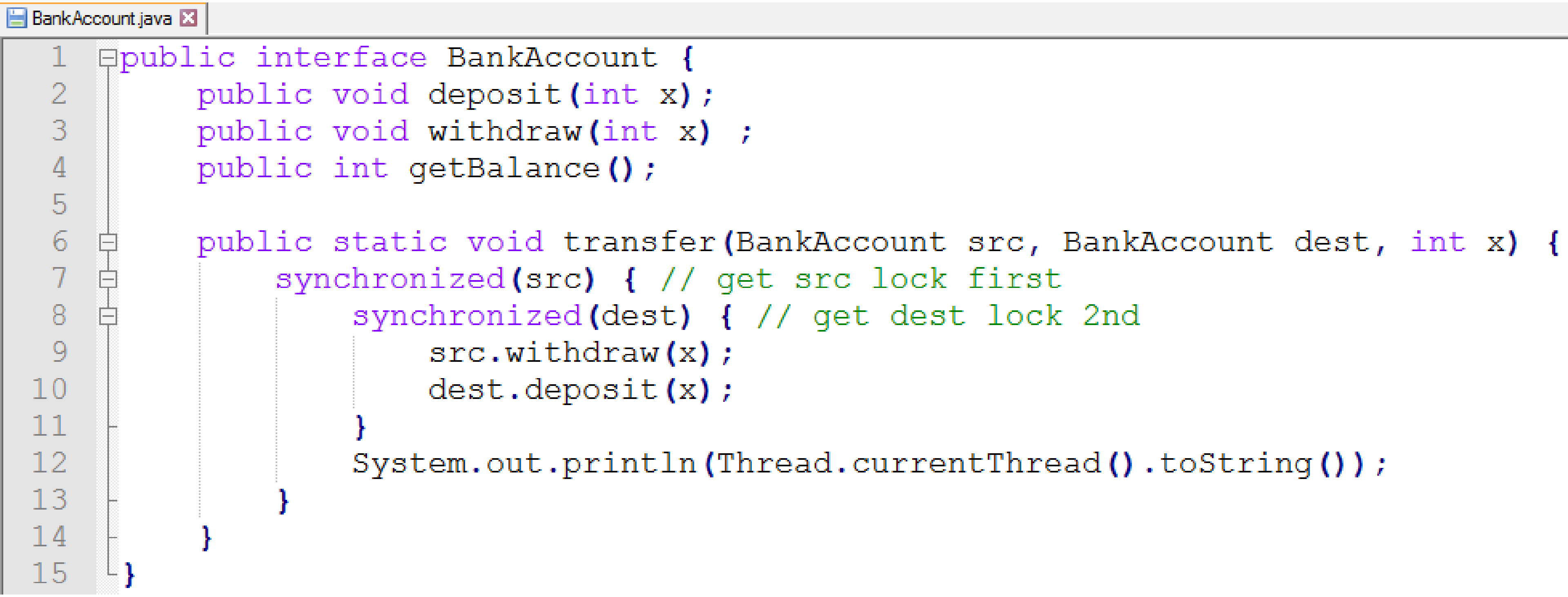
- Inconsistent lock ordering causes deadlock

Contention Over 2 Locks

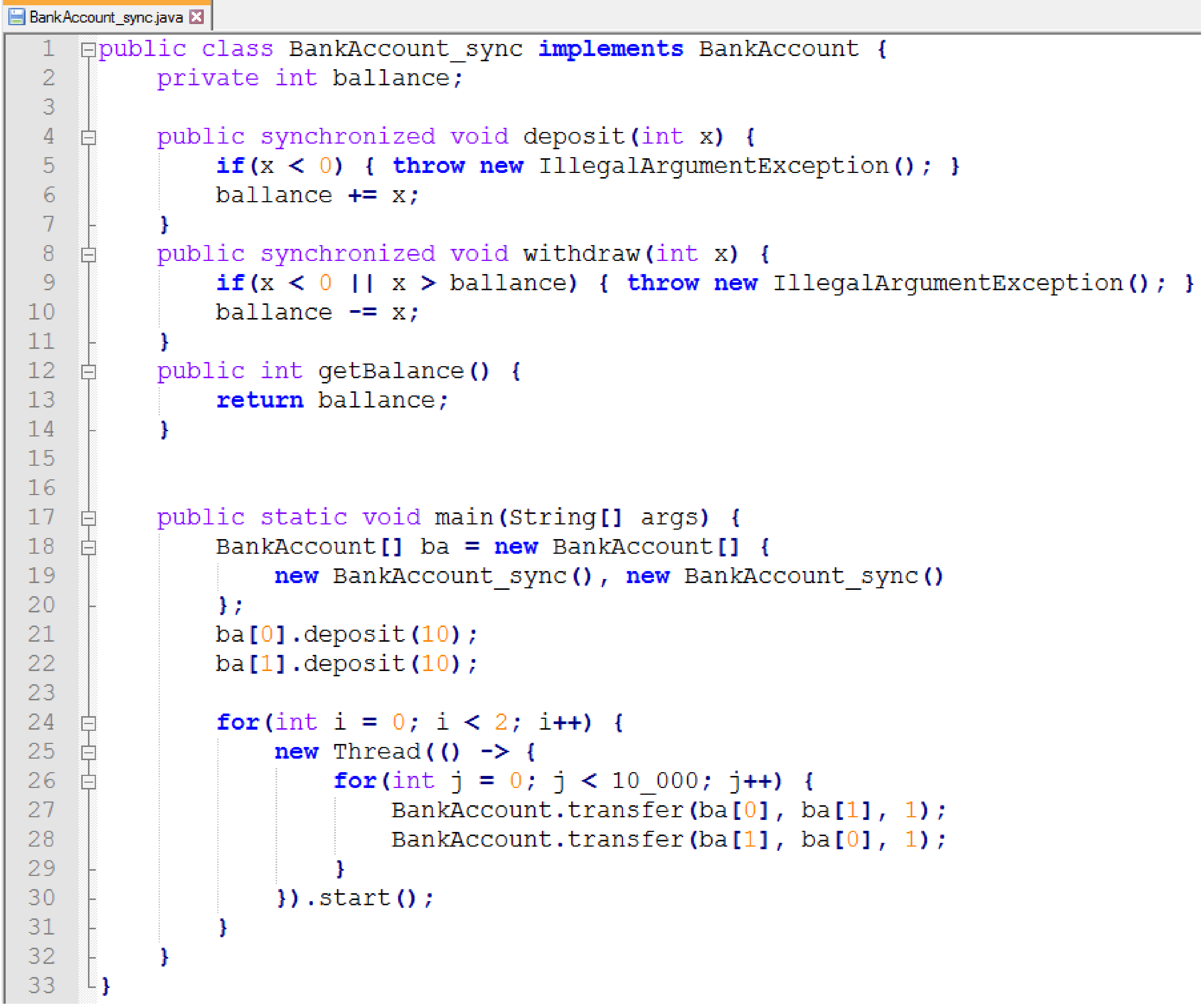


```java
public class FriendDeadlock {
  private final String name;
  public FriendDeadlock(String name) {
    this.name = name;
  }
  public synchronized void bow(FriendDeadlock bower) {
    System.out.println(name + " has bowed to " + bower.name);
    bower.bowBack(this);
  }

  public synchronized void bowBack(FriendDeadlock bower) {
    System.out.println(name + " has bowed back to " + bower.name);
  }

  public static void main(String[] args) {
    final FriendDeadlock alice = new FriendDeadlock("Alice");
    final FriendDeadlock bob = new FriendDeadlock("Bob");
    new Thread(() -> {alice.bow(bob);}).start();
    new Thread(() -> {bob.bow(alice);}).start();
  }
}
```


T1
alice mutex
bob mutex
T2
alice synch
acquire
Held by T1
Held by T2
acquire
bob synch
alice bow
bob bow
bob synch
acquire
acquire
alice synch
deadlock
bob bowBack
release
release
alice bowBack
release
release



```java
public interface BankAccount {
    public void deposit(int x);
    public void withdraw(int x) ;
    public int getBalance();

    public static void transfer(BankAccount src, BankAccount dest, int x) {
        synchronized(src) { // get src lock first
            synchronized(dest) { // get dest lock 2nd
                src.withdraw(x);
                dest.deposit(x);
            }
            System.out.println(Thread.currentThread().toString());
        }
    }
}
```



```java
public class BankAccount_sync implements BankAccount {
    private int ballance;

    public synchronized void deposit(int x) {
        if(x < 0) { throw new IllegalArgumentException(); }
        ballance += x;
    }
    public synchronized void withdraw(int x) {
        if(x < 0 || x > ballance) { throw new IllegalArgumentException(); }
        ballance -= x;
    }
    public int getBalance() {
        return ballance;
    }


    public static void main(String[] args) {
        BankAccount[] ba = new BankAccount[] {
            new BankAccount_sync(), new BankAccount_sync()
        };
        ba[0].deposit(10);
        ba[1].deposit(10);

        for(int i = 0; i < 2; i++) {
            new Thread(() -> {
                for(int j = 0; j < 10_000; j++) {
                    BankAccount.transfer(ba[0], ba[1], 1);
                    BankAccount.transfer(ba[1], ba[0], 1);
                }
            }).start();
        }
    }
}
```

Livelock

- **Def'n**: ***Livelock*** occurs under similar conditions as deadlock, except that
  - the threads continuously change their states in response to one another
  - none can make progress because they are too busy responding to each other to do any work
  - Eg: 2 people attempting to pass each other in a hallway

Starvation

- **Def'n**: ***starvation*** occurs when a thread is continually denied access to a resource that it needs to perform its work.
- Eg:
  - A system has 2 priorities for threads: high and low
  - The high priority threads have enough work to use all of the resources all of the time
  - The low priority tasks never get access to the resources
- Solution: aging
  - A process gains higher priority the longer it waits

Priority Inversion

- A possible consequence of starvation
- Eg: Suppose 3 priority levels: high, mid, low
  - Suppose a high priority task depends on the result of a low priority task
  - Then the high priority task only advances when a low priority task advances
    - Effectively giving the high priority task the same priority as the low priority task
  - The high priority task can be starved if the low priority tasks are starved by the mid priority tasks

Recap

- Deadlock
  - 1 lock
  - Condition variables
  - 2 locks
  - Livelock
  - Starvation
  - Priority Inversion