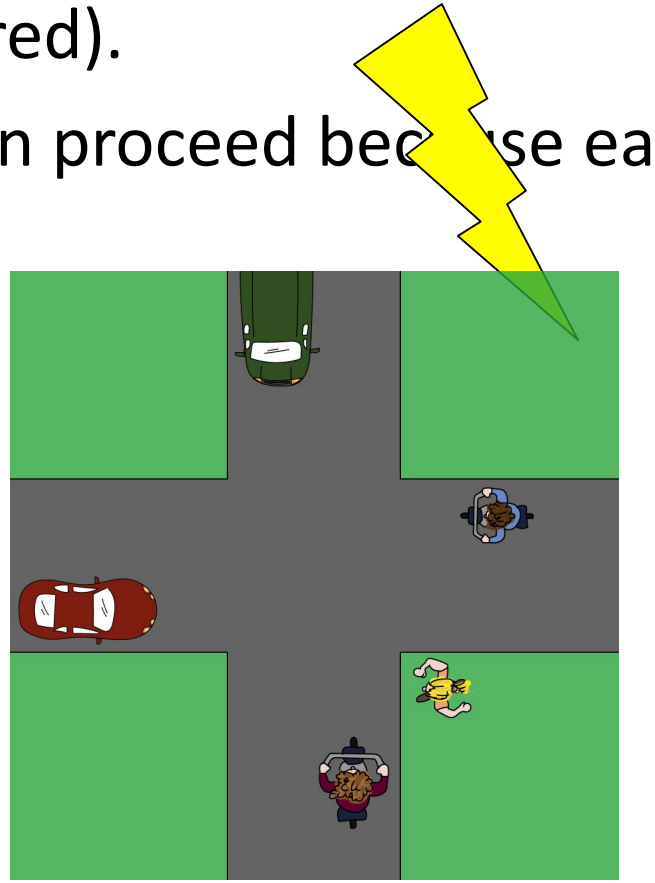
Concurrency (III)

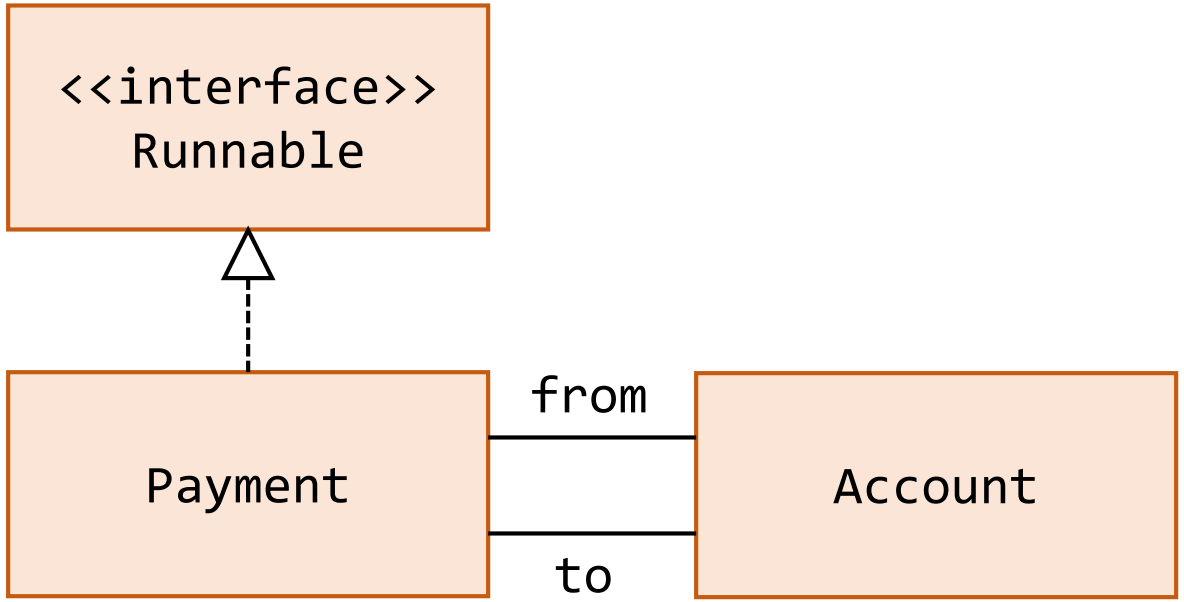
Lecture 14 (June 1st, 2021)

Termination or Deadlock

- Termination: all (non-deamon) tasks have been completed (either normally or an exception has occurred).
- Deadlock: No thread can proceed because each thread is waiting for another to do some work first.

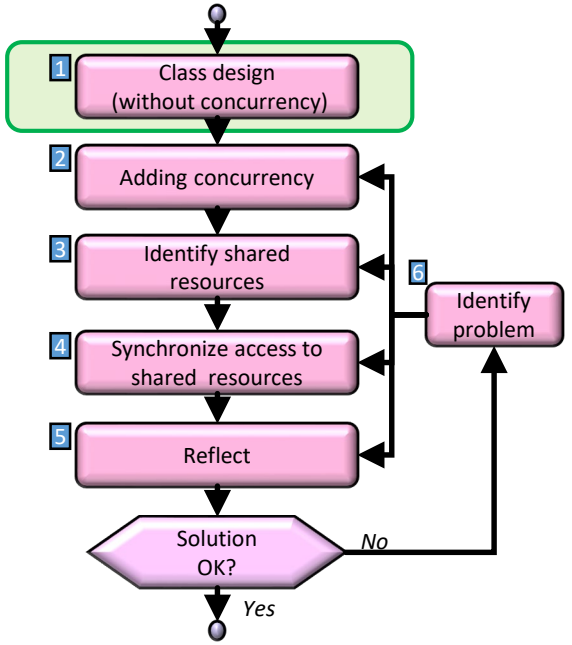


Starting point



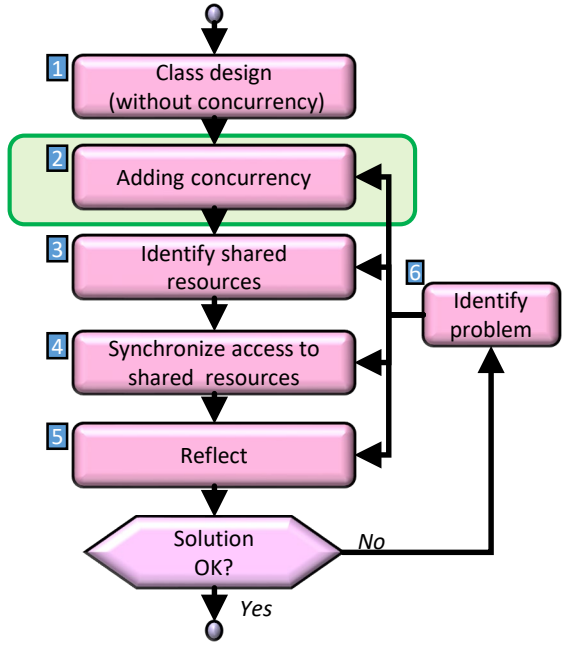
Example: class Account

```
public class Account {
    private final int myAccountNumber;
    private int       myBalance;

    public Account( int accountNumber, int initialBalance ) {
        this.myAccountNumber = accountNumber;
        this.myBalance       = initialBalance;
    }

    public void deposit( int amount ) {
        myBalance += amount;
    }

    public void withdraw( int amount ) {
        myBalance -= amount;
    }

    public String toString() {
        return String.format("%d: %d\n", myAccountNumber, myBalance );
    }
}
```



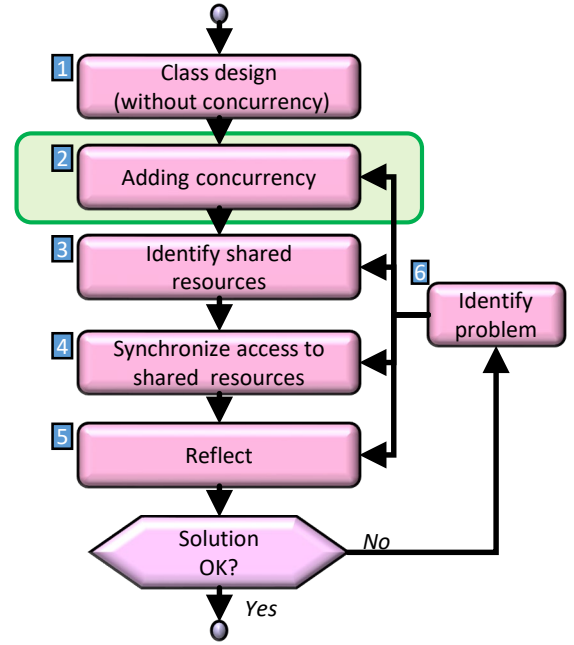
Example: Bank transfer

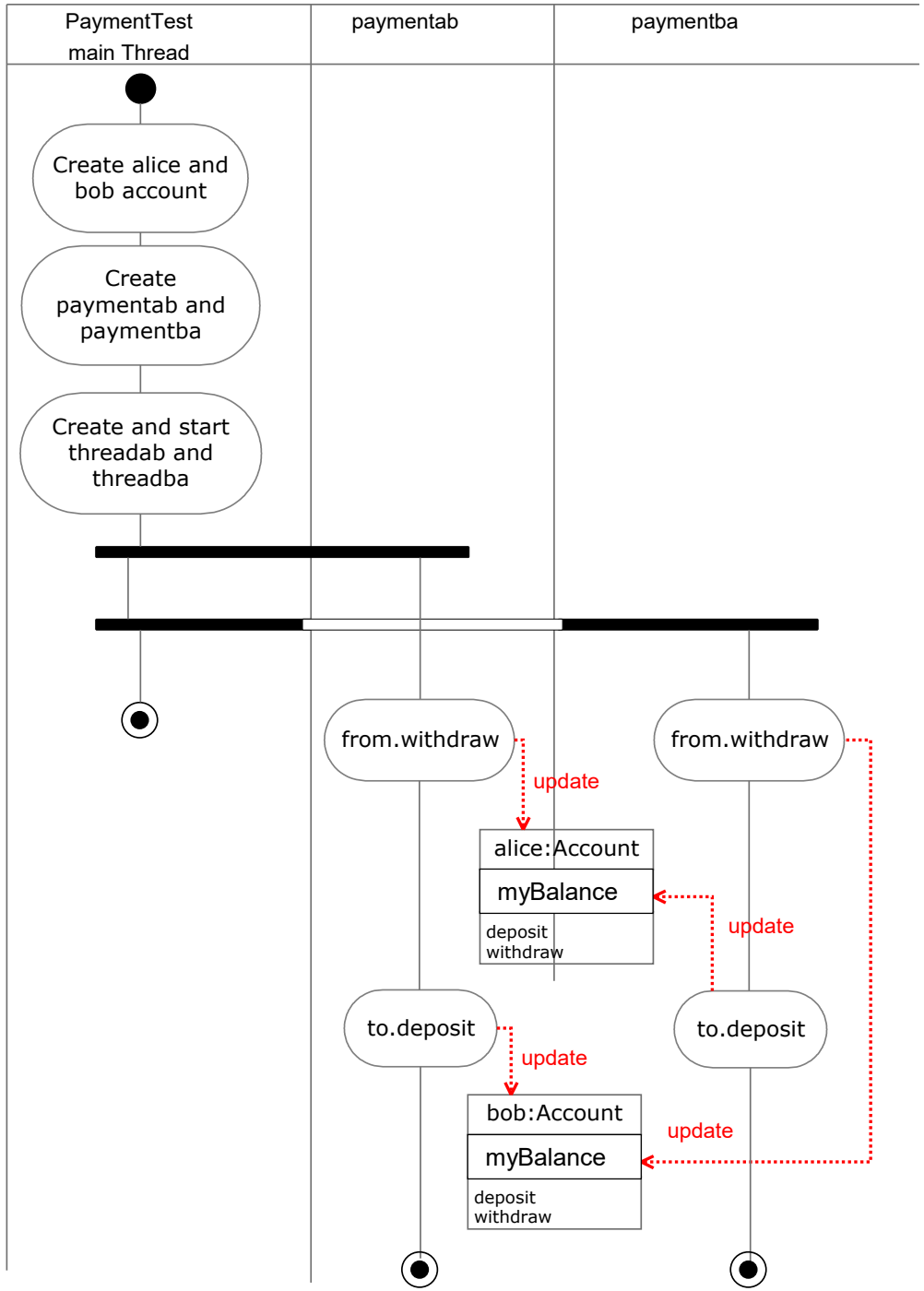
```
public class Payment implements Runnable {
  private final Account from, to;
  private final int amount;

  public Payment( Account from, Account to, int amount ){
     this.from   = from;
     this.to     = to;
     this.amount = amount;
  }

  public void run() {
    from.withdraw( amount );
    to.deposit( amount );
    System.out.println( from );
    System.out.println( to );
  }
}
```

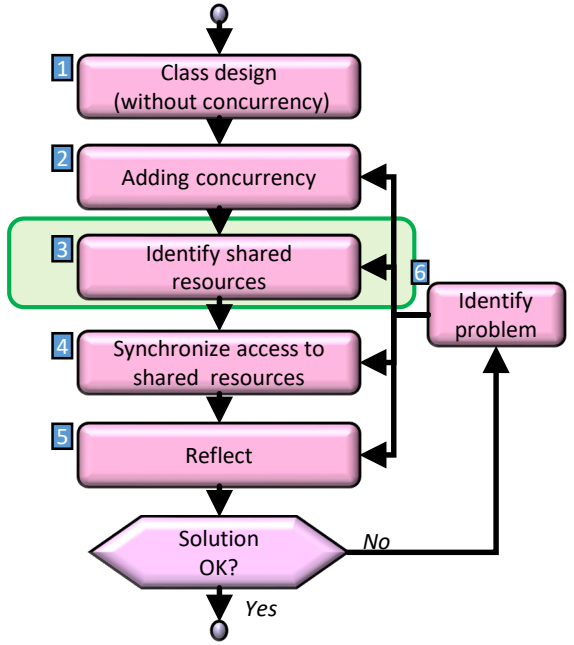


Test this class

```
public class PaymentTest {

    public static void main( String[] args ) {
        Account alice = new Account(20140001, 100);
        Account bob   = new Account(20140002, 100);

        ExecutorService executor = Executors.newCachedThreadPool();

        Payment paymentab = new Payment(alice, bob, 20);
        Payment paymentba = new Payment(bob, alice, 30);

        executor.execute(paymentab);
        executor.execute(paymentba);
        executor.shutdown();
    }
}
```





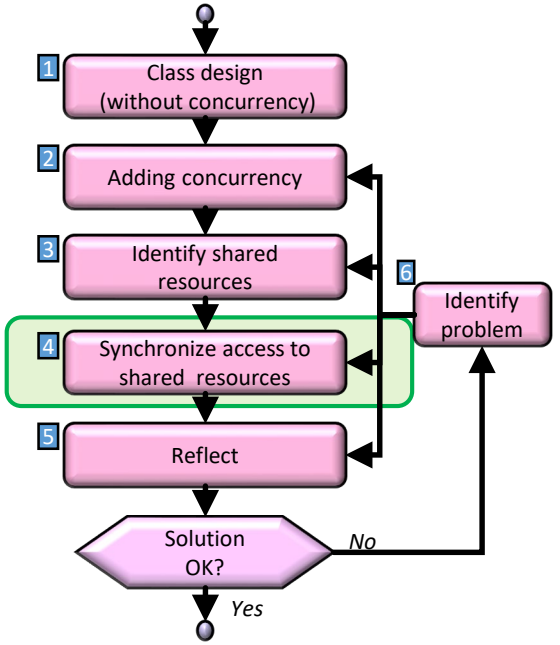


Analysis: Race conditions

Attention: we now are dealing with two different shared objects that are accessed from both threads.

Synchronization is necessary

- Incorrect amounts can result
  - because `withdraw` en `deposit` are not atomic
  - solved by locking critical sections of `Account`
  - recommended method for using locks:




```
lock.lock();
try {
  // access to the shared resource
} finally {
   lock.unlock();
}
```

critical section

recall: both setters and getters need to be sychronized

class Account synchronized

```
public class Account {
   private final int myAccountNumber;
   private int myBalance;
   private Lock lock = new ReentrantLock();

   public Account( int accountNumber,
                   int initialBalance ) {
      this.myAccountNumber = accountNumber;
      this.myBalance       = initialBalance;
   }

   public int getAccountNumber() {
      return myAccountNumber;
   }
   ...
}
```

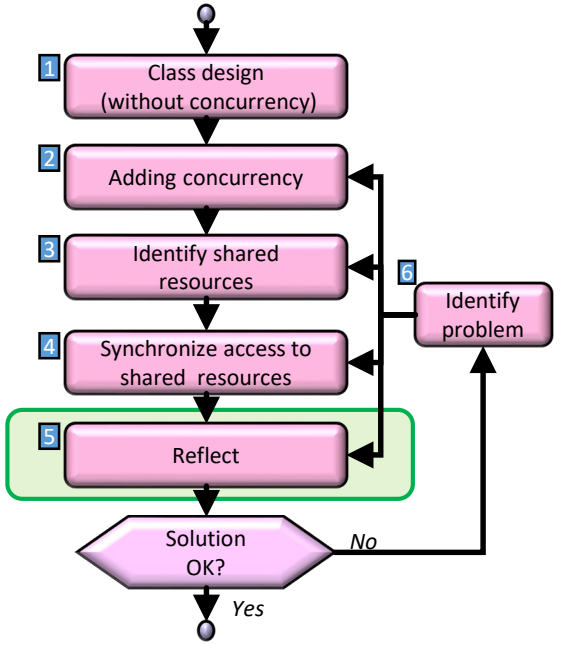
```
public static class InsufficientFundException
                              extends Exception {
   public InsufficientFundException() {
      super( "Insufficient balance" );
   }
}
```

```
public void deposit( int amount ) {
   lock.lock();
   try {
      myBalance += amount;
   } finally {
      lock.unlock();
   }
}

public void withdraw( int amount )
            throws InsufficientFundException {
  lock.lock();
  try {
     if ( myBalance < amount ) {
        throw new InsufficientFundException();
     }
     myBalance -= amount;
  } finally {
     lock.unlock();
  }
}
```

Output (2 example runs)

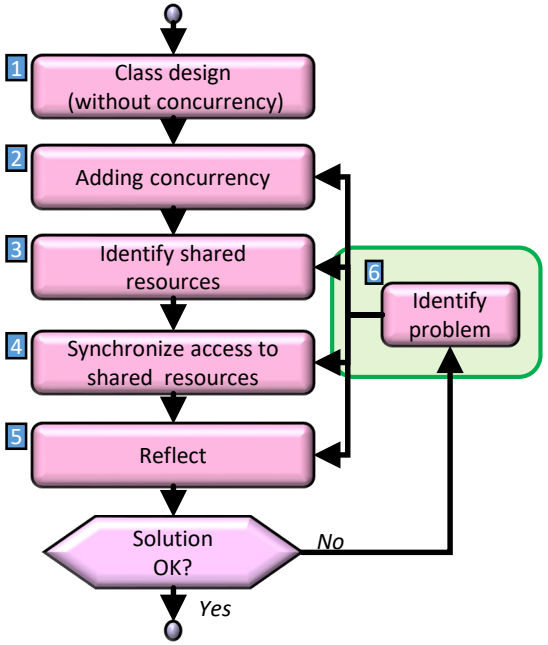
```
20140001:80
20140002:120
20140002:90
20140001:110
```

```
20140001:80
20140002:90
20140002:90
20140001:110
```



Simple synchronization insufficient

- Inconsistent states are still possible
  - Explanation: several withdrawals from `from` have occured that have not yet been processessed as deposits to `to`.
  - Outputting the balances gives a total that is too low because there are amounts still “on their way” that are not taken into account.
- Solution: What we need here is the assurance that the withdraw- and deposit-actions are always executed in pairs.

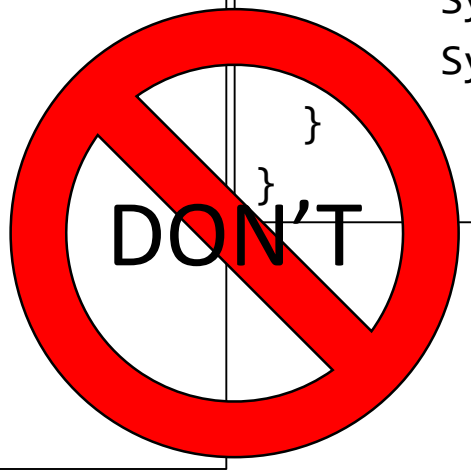


Solution (first attempt)

- Extend class Account with a transfer method

```
public void transfer( int amount, Account to )
                        throws InsufficientFundException {
   lock.lock();
   to.lock.lock();
   try {
      withdraw( amount );
      to.deposit(amount);
   } finally {
      lock.unlock();
      to.lock.unlock();
   }
}
```

```
public class Payment implements Runnable {
    …
    public void run() {
        from.transfer( amount, to );
        System.out.println( from );
        System.out.println( to );
    }
}
```

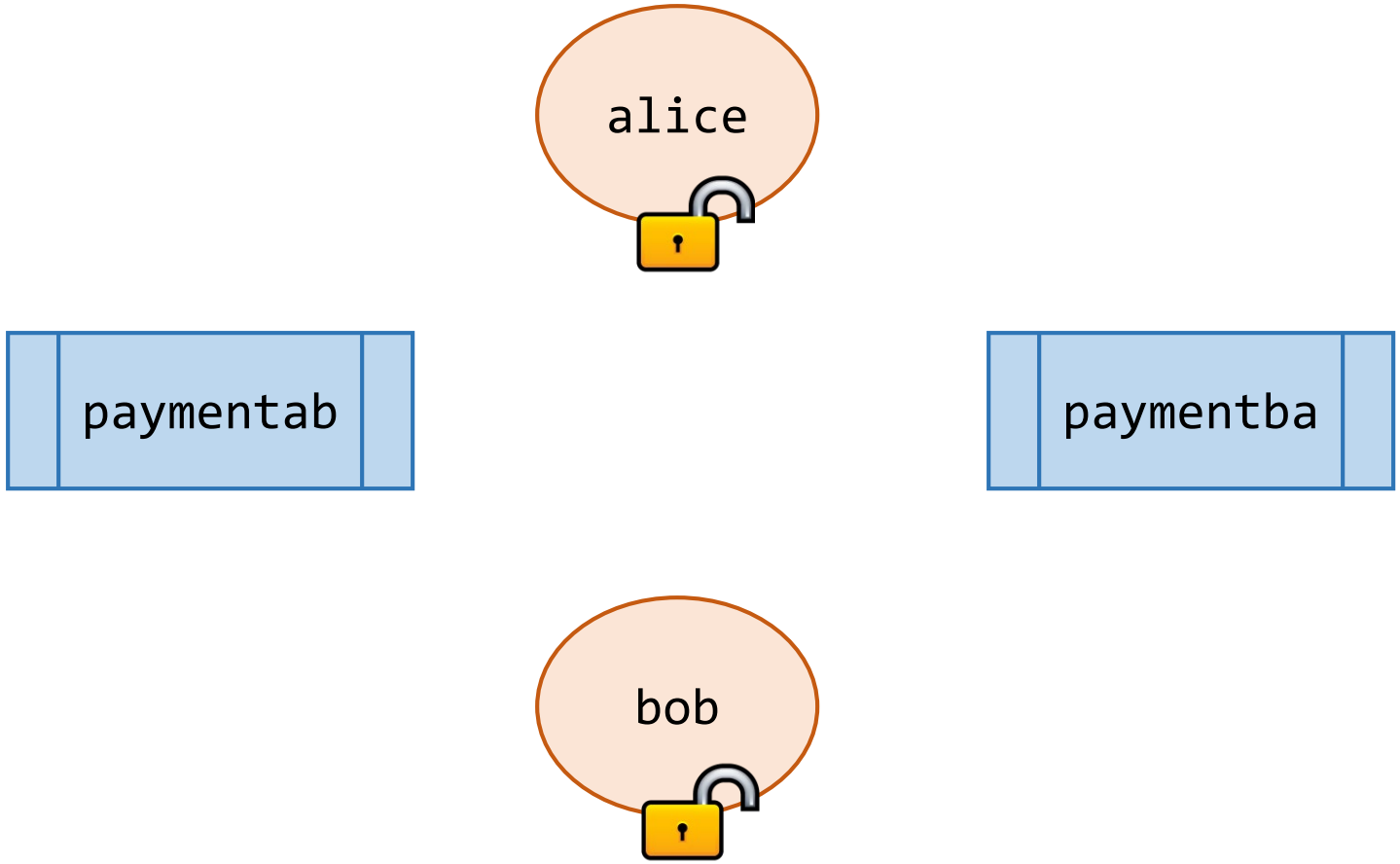


- Deadlock: a program execution that appears to neither make progress nor terminate

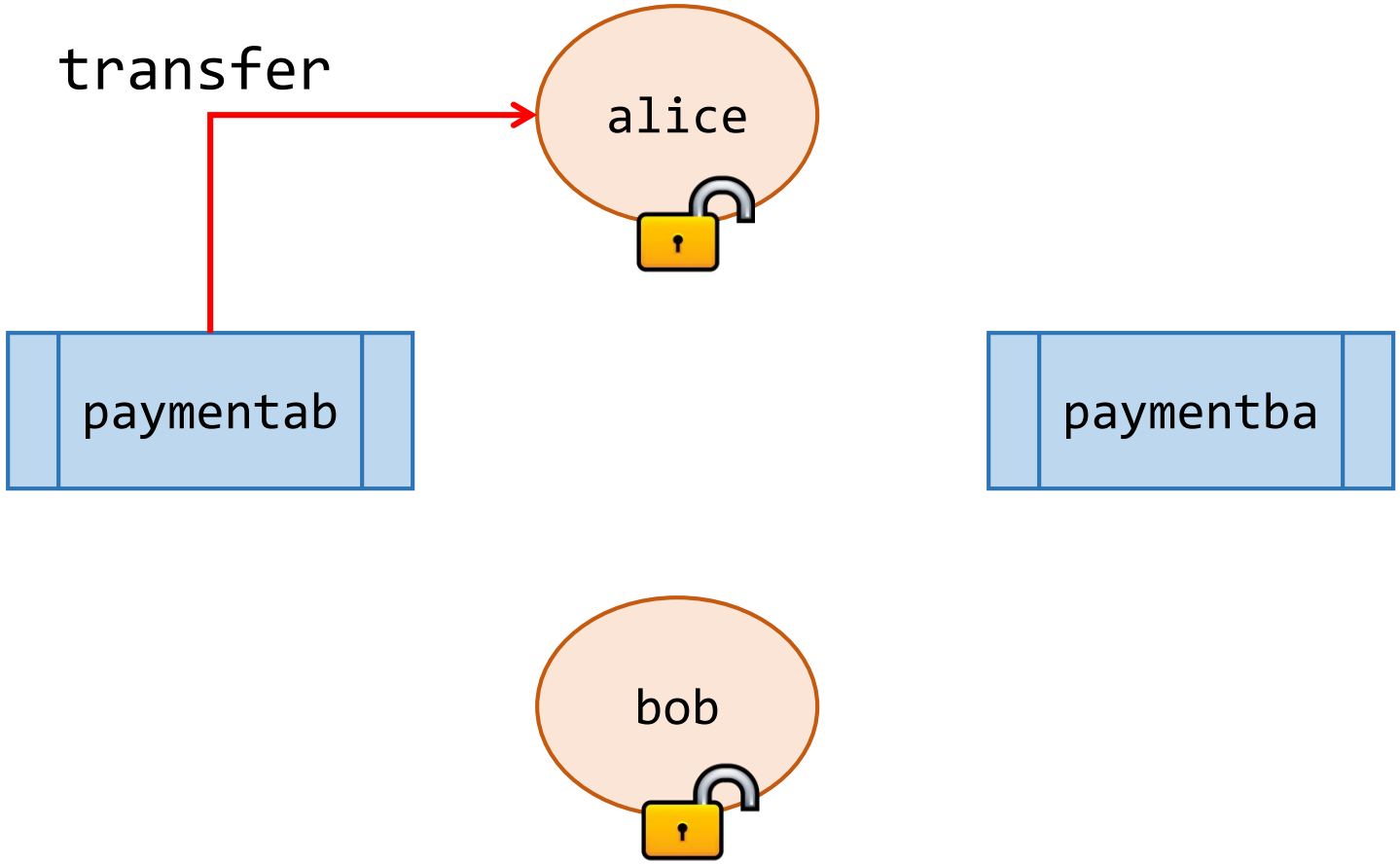
The reason

- Scenario that leads to deadlock:
  - `threadab` executes `alice.transfer` and thus acquires lock `alice`
  - `threadab` now owns `alice`'s lock
  - `threadba` executes `bob.transfer` and thus acquires lock `bob`
  - `threadba` now owns `bob`'s lock
  - `threadab` executes `bob.lock.lock()`and waits for the release of `bob`'s lock
  - `threadba` executes `alice.lock.lock()` and waits for the release of `alice`'s lock
  - Both threads now wait for one another

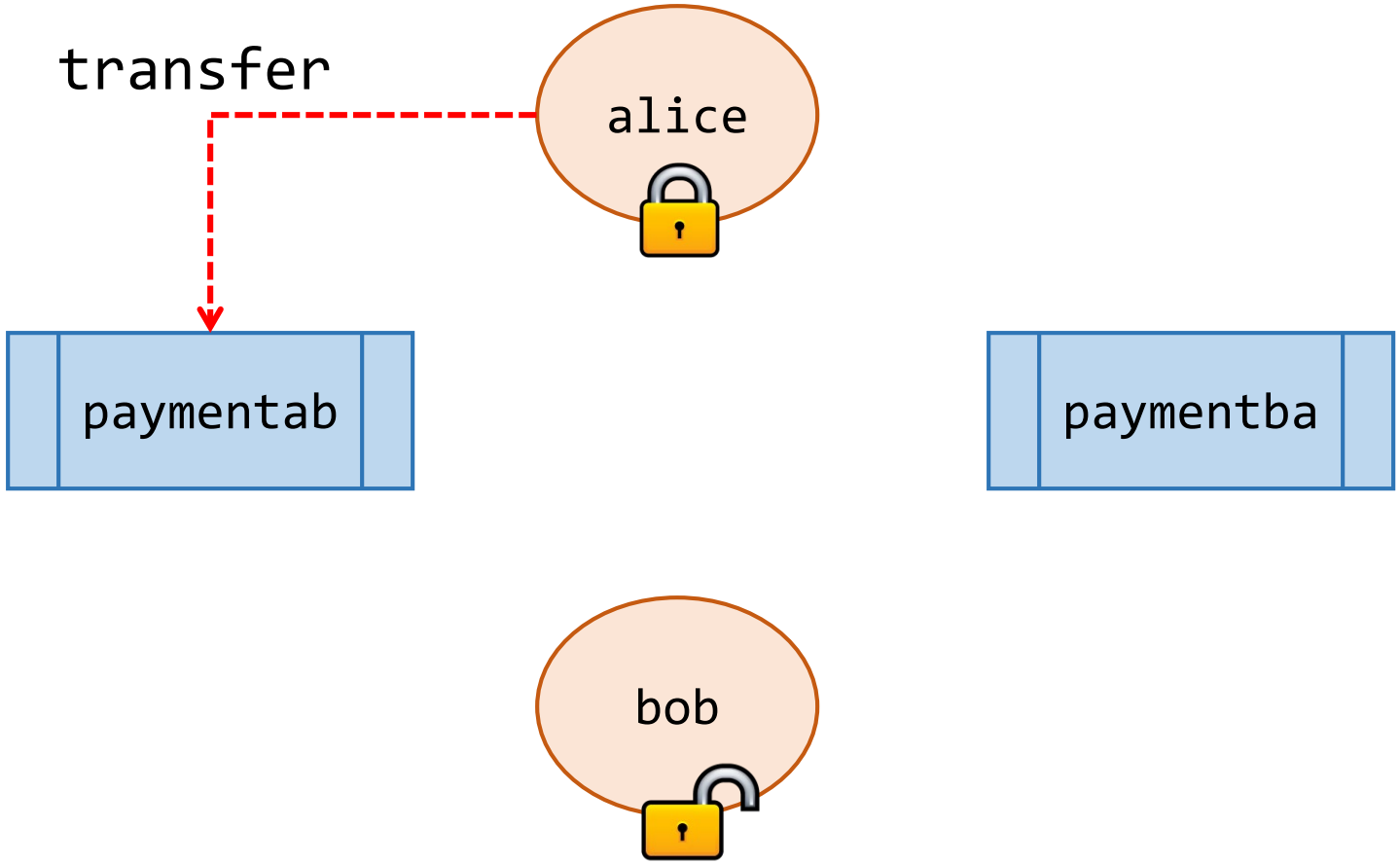
Communication diagram



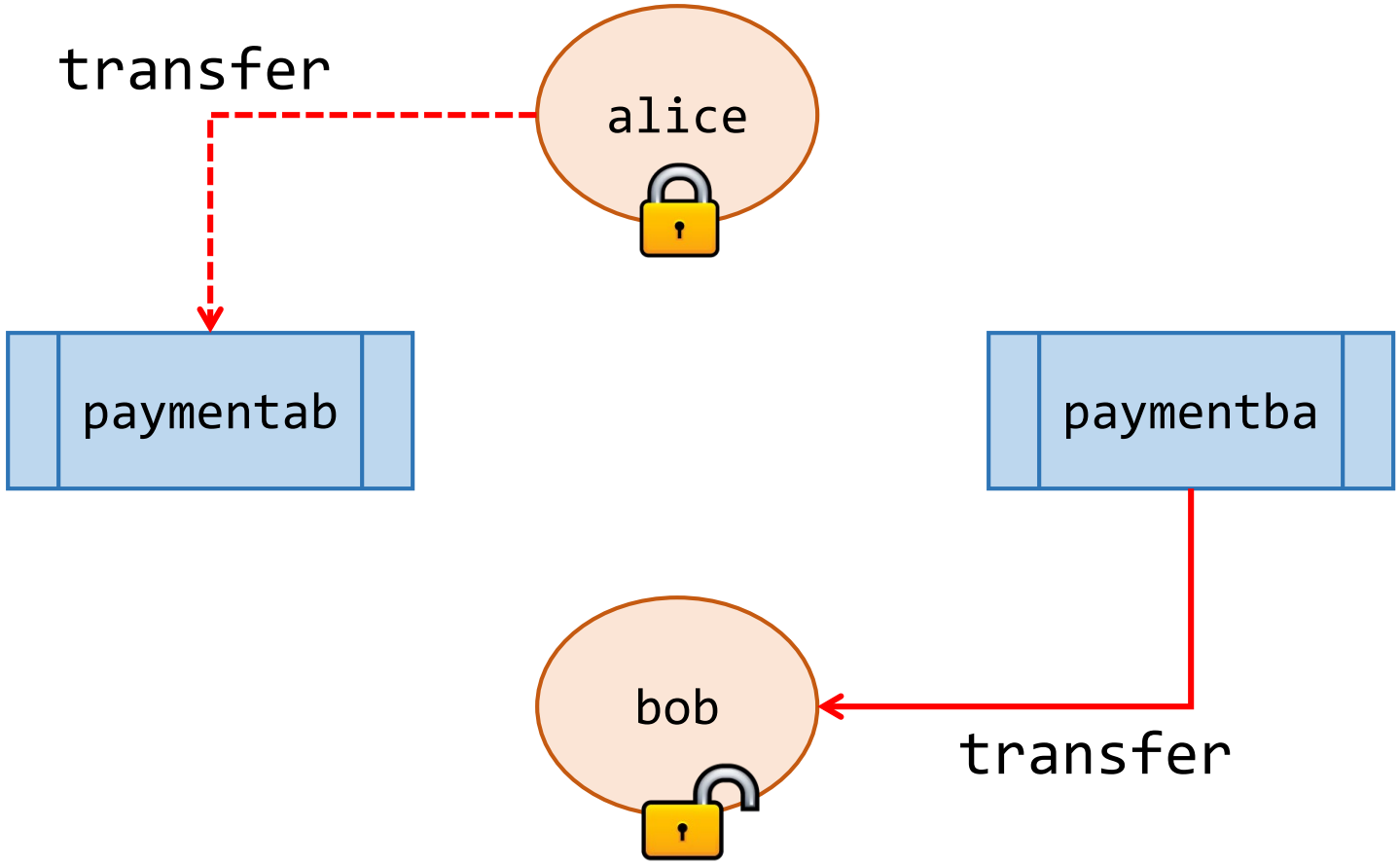
Communication diagram



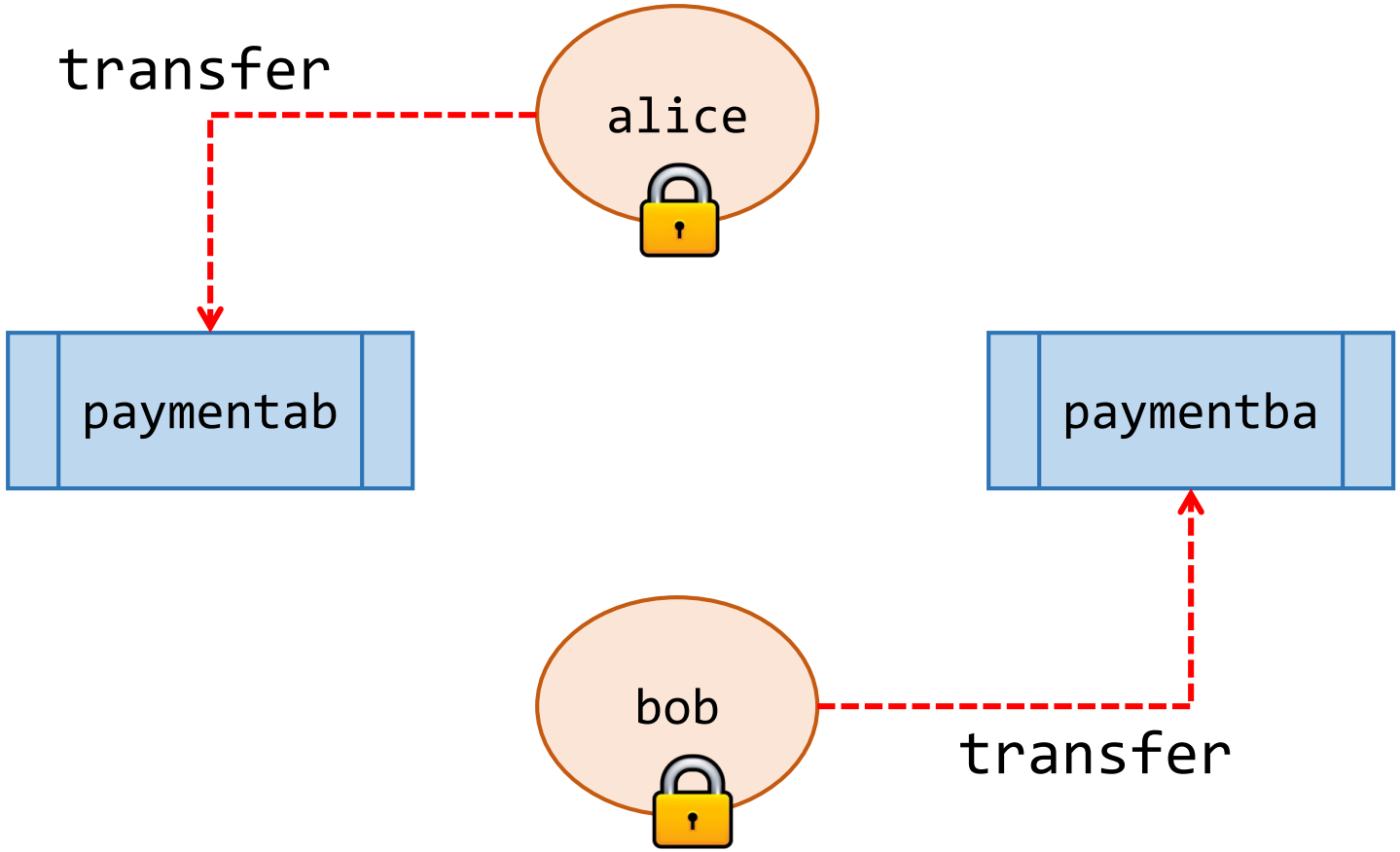
Communication diagram



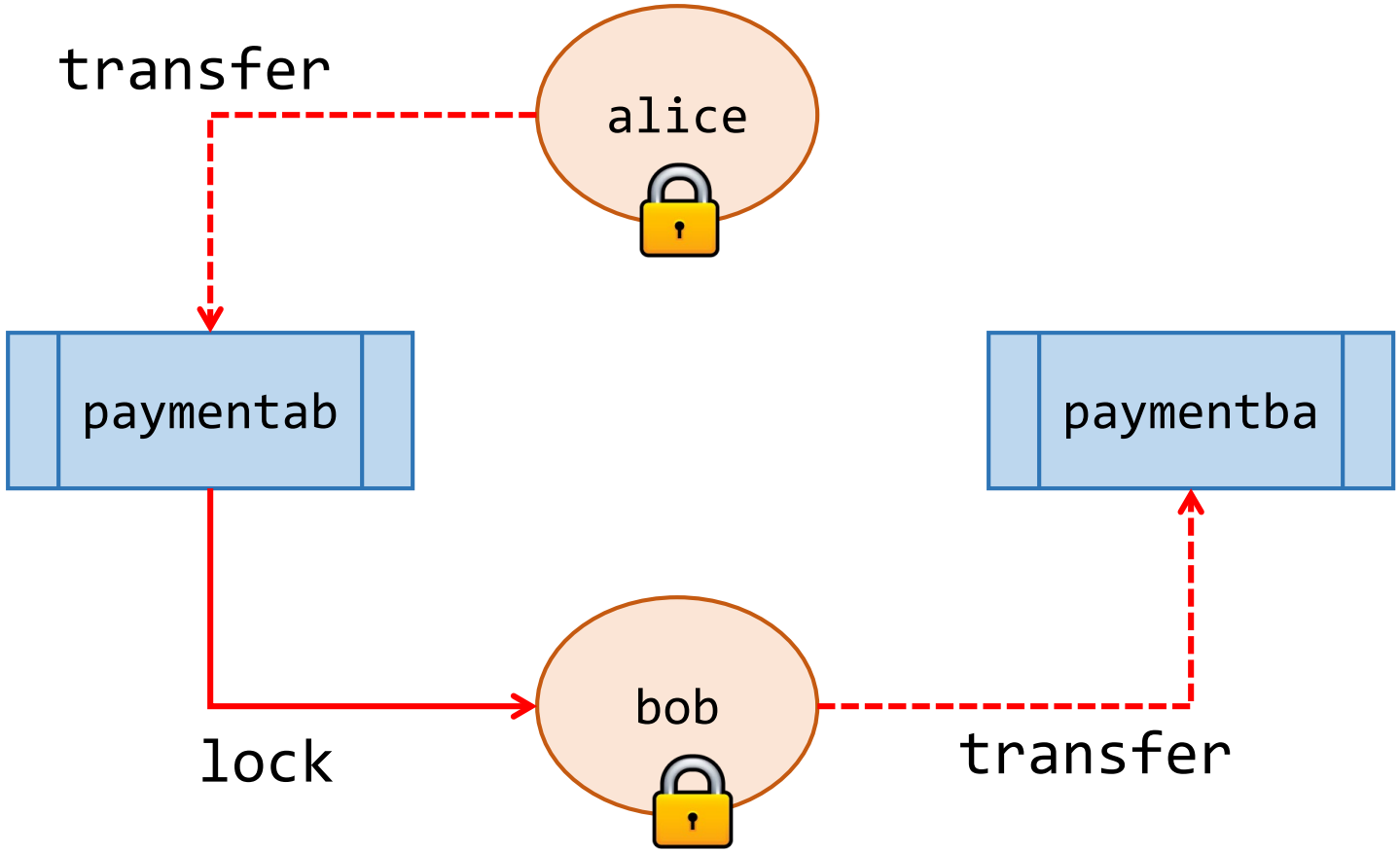
Communication diagram



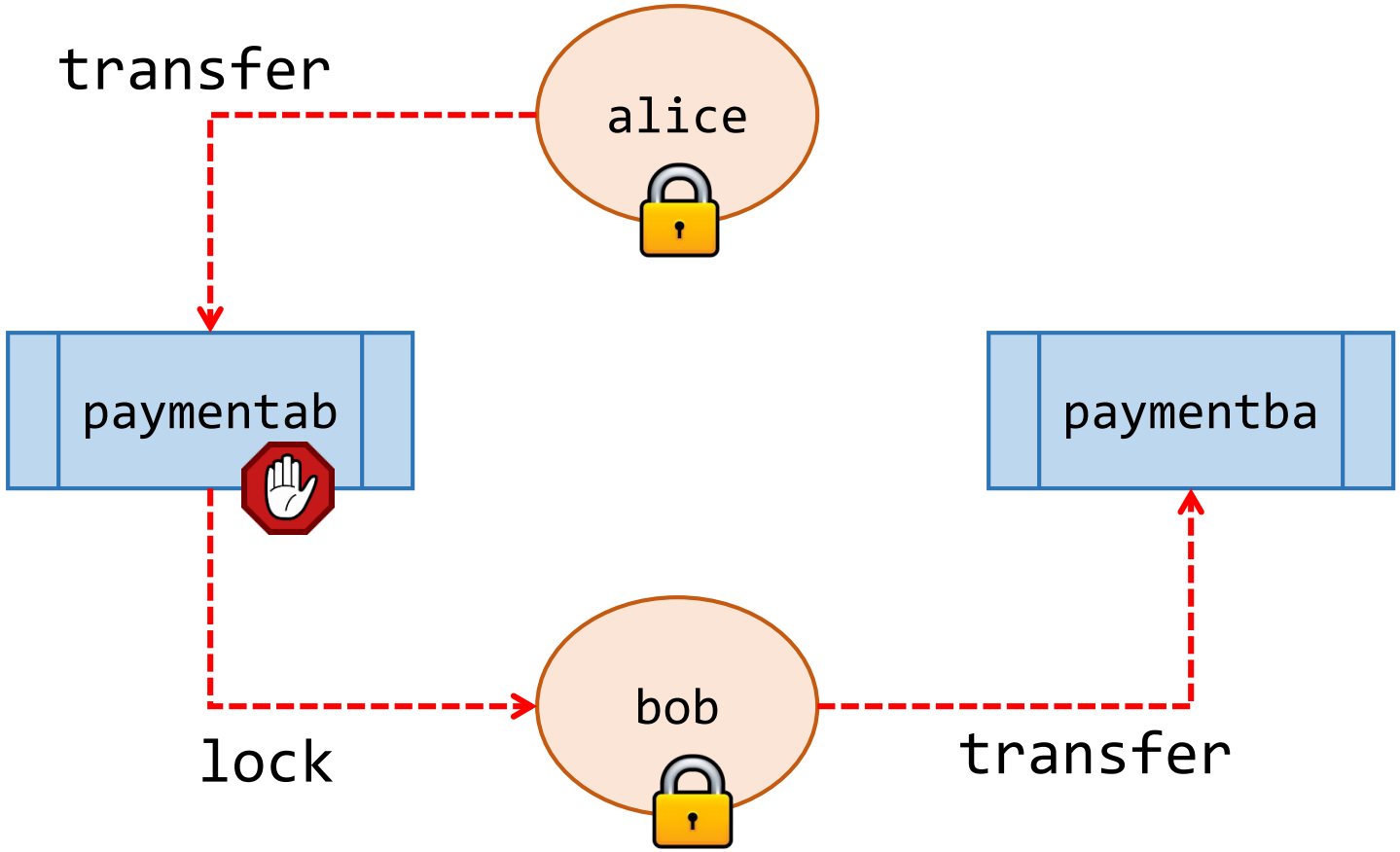
Communication diagram



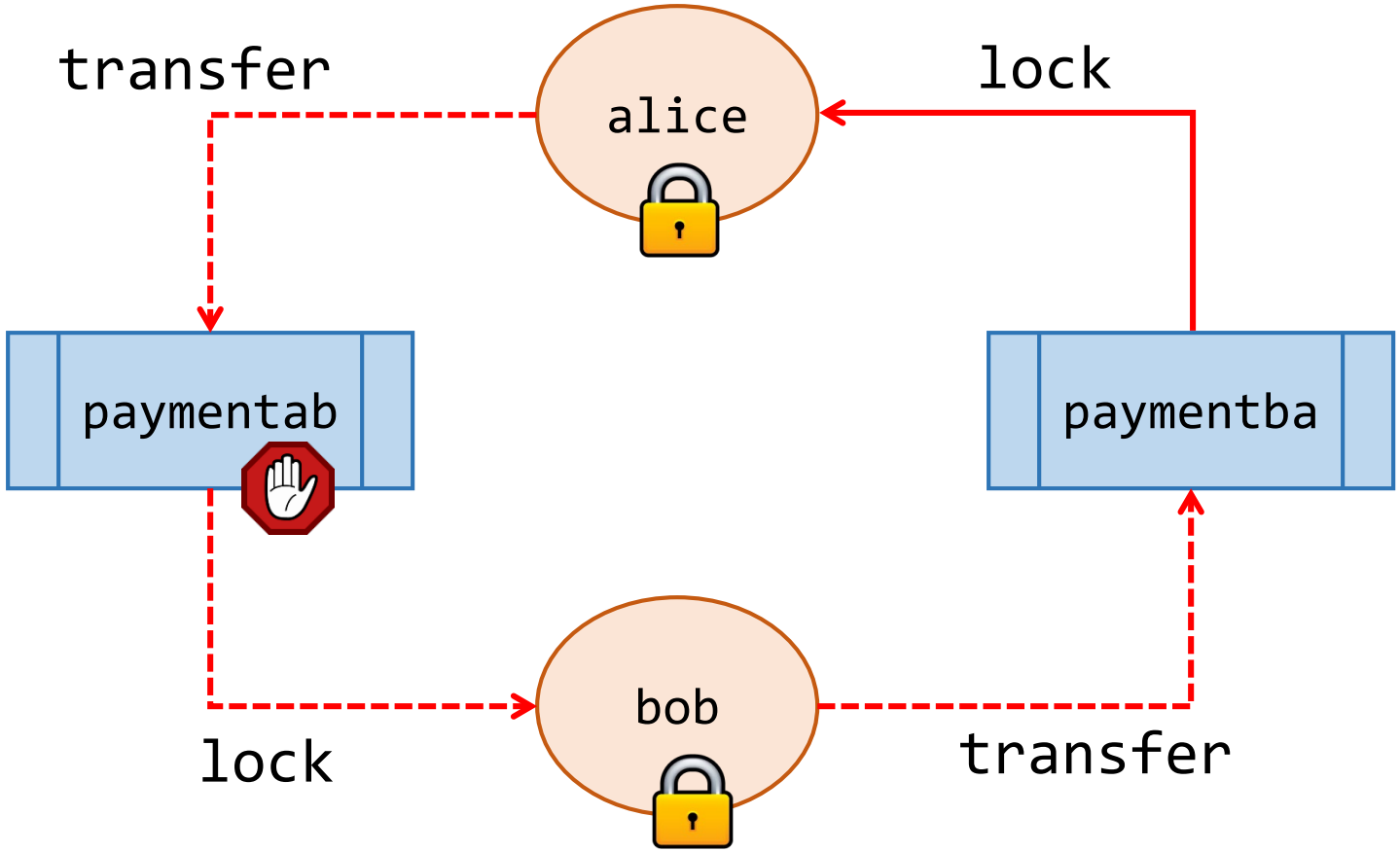
Communication diagram



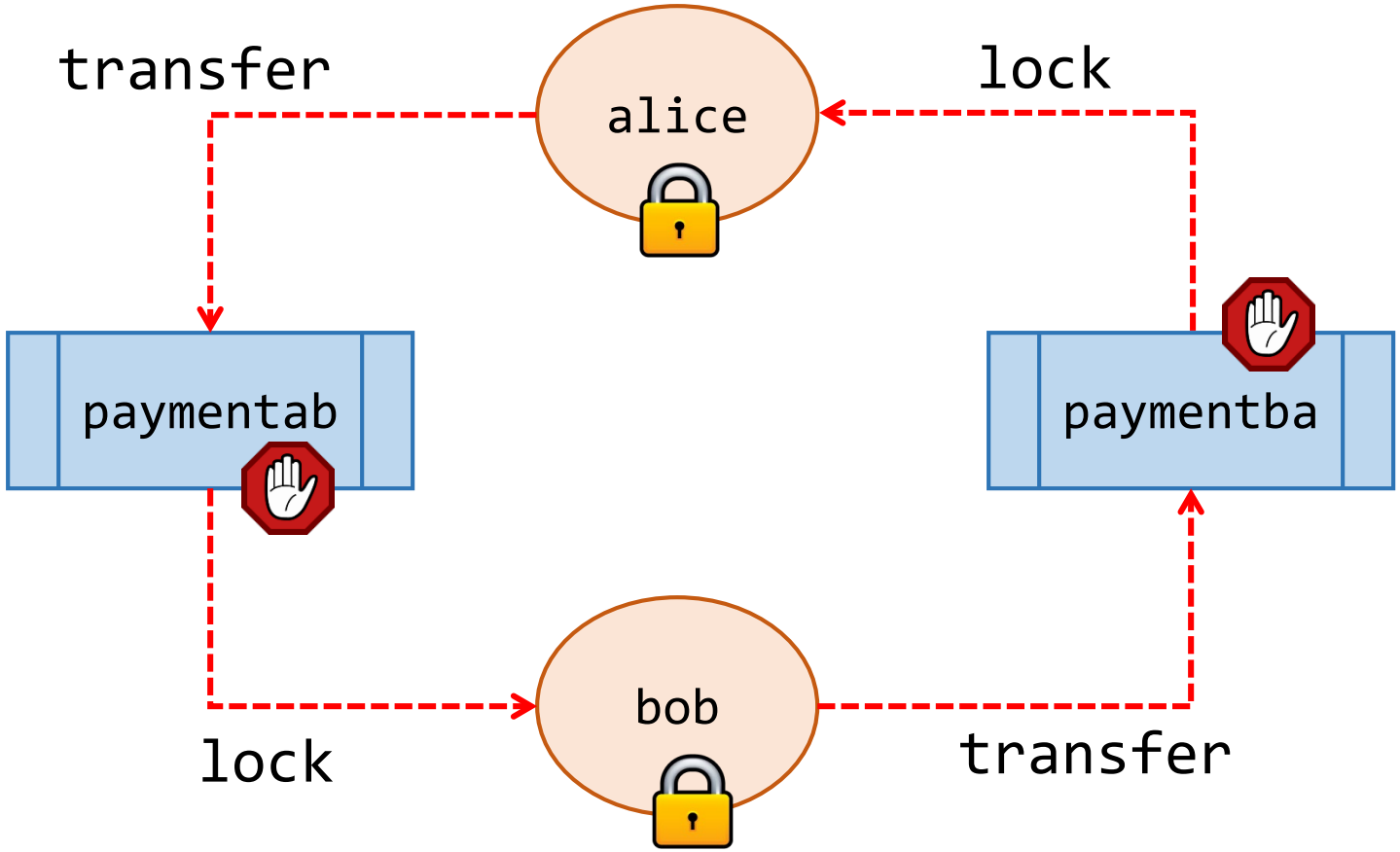
Communication diagram



Communication diagram



Communication diagram



Solution: use same locks always in the same order.

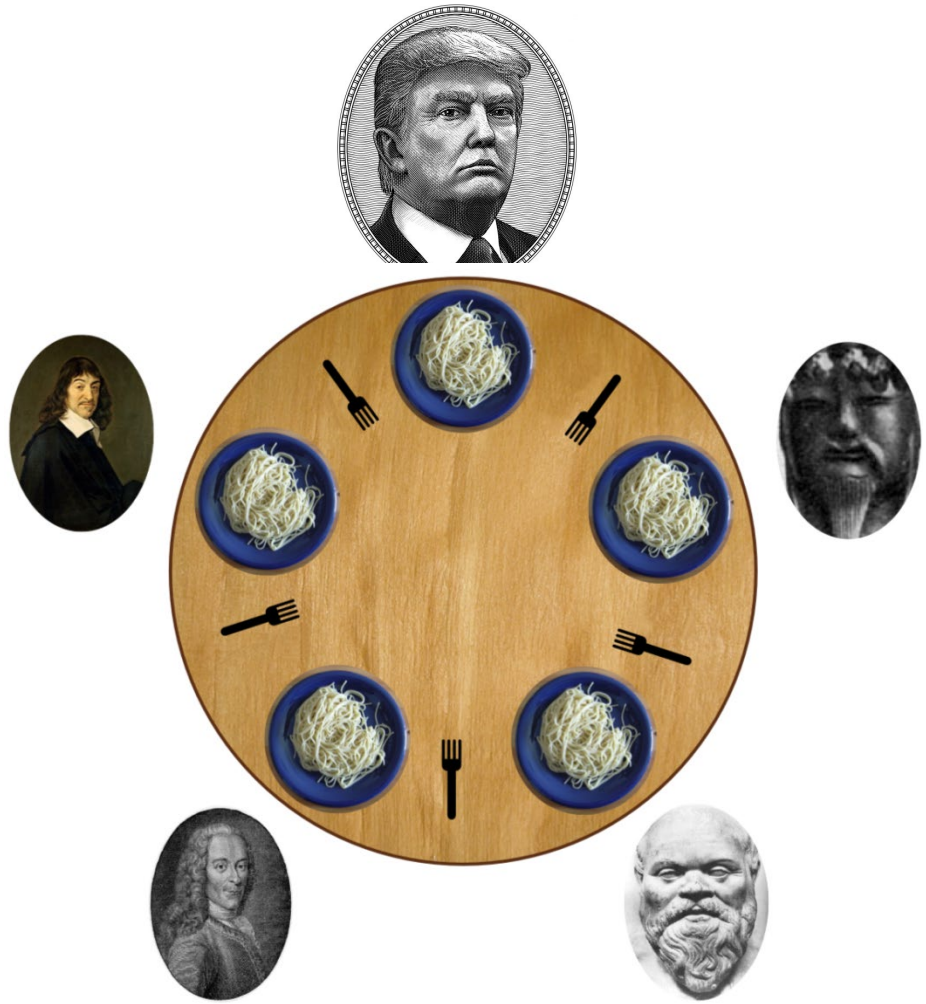
```
public void transfer( int amount, Account to ) throws InsufficientFundException {
    Lock first, second;
    if ( myAccountNumber < to.myAccountNumber ) {
       first = myLock; second = to.myLock;
    } else {
       first = to.myLock; second = myLock;
    }
    first.lock();
    try {
        second.lock();
        try {
            withdraw( amount );
            to.deposit(amount);
        } finally {
            second.unlock();
        }
    } finally {
        first.unlock();
    }
}
```

requires a global ordering on locks

here we use the ordering on account numbers

Dining Philosophers (Dijkstra)

- N philosophers and N forks
- Philosophers eat/think
- Eating needs 2 forks
- Pick one fork at a time



Dining Philosophers: class **Fork**

```
public class Fork {
   private boolean iAmPickedUp = false;
   private Lock myLock = new ReentrantLock();
   private Condition forkPutBack = myLock.newCondition();

   public void pickUp() throws InterruptedException {
      myLock.lock();
      try {
         while ( iAmPickedUp ) {
            forkPutBack.await();
         }
         iAmPickedUp = true;
      } finally {
         myLock.unlock();
      }
   }
   <...>
}
```

```
public class Fork {
    <...>

    public void putDown () {
        myLock.lock();
        try {
            iAmPickedUp = false;
            forkPutBack.signalAll();
        } finally {
            myLock.unlock();
        }
    }
}
```

Dining Philosophers: class `Philosopher`

```
public class Philosopher
             implements Runnable {
  private Fork myLeftFork;
  private Fork myRightFork;
  private int myId;
  public Philosopher( Fork left,
              Fork right, int id ) {
     myLeftFork  = left;
     myRightFork = right;
     myId        = id;
  }

  private void doAction(String action){
    System.out.println( "Philosopher "
        + myId + action );
    takeABreak(
      (int) (Math.random() * 100) );
  }
```

```
@Override
public void run() {
   while ( true ) {
      try {
         doAction("Thinking");
         myLeftFork.pickUp();
         doAction("Picked up left fork");
         myRightFork.pickUp();
         doAction("Picked up right fork - now eating");
         myLeftFork.putDown();
         doAction("Put down left fork");
         myRightFork.putDown();
         doAction("Put down left fork - back to thinking");
      } catch (InterruptedException e) {
          System.out.println("Unexpected interrupt");
          System.exit(0);
      }
   }
}
```

Radboud University

Dining Philosophers: class `DiningPhilosophers`

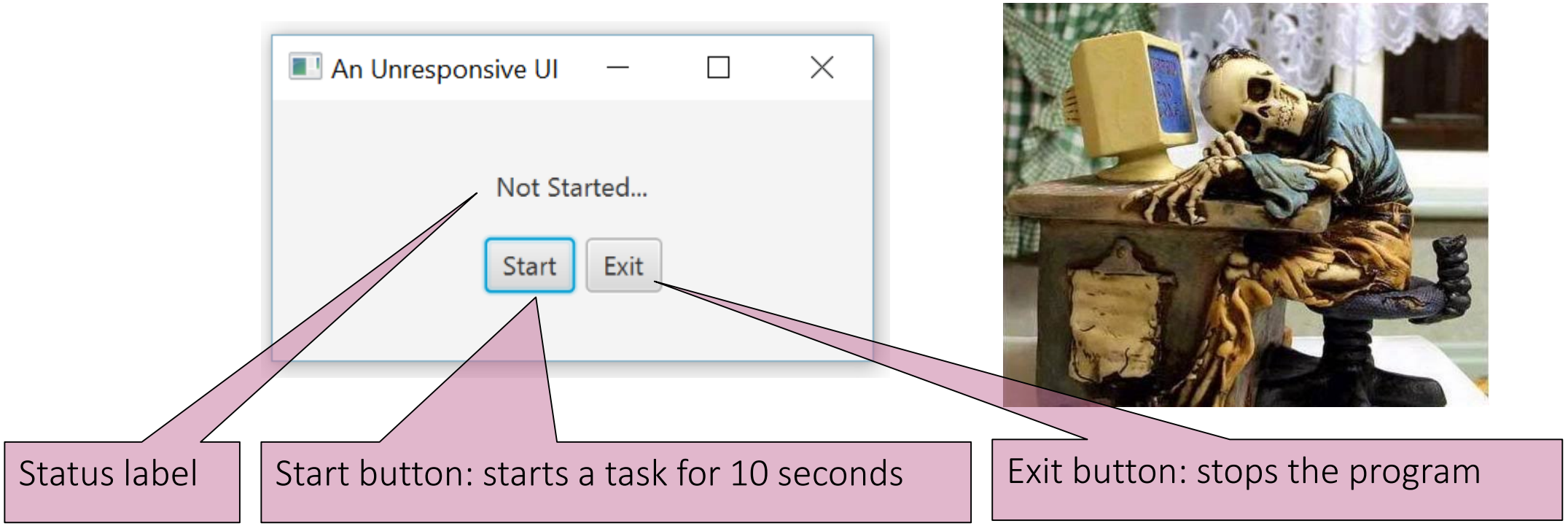
```
public class  DiningPhilosophers {
  private static final int NR_PHILOSOPHERS = 5;

  public void start() {
    Fork[] forks  = new Fork [NR_PHILOSOPHERS];
    for ( int i = 0; i < NR_PHILOSOPHERS; i++ ) {
        forks[i] = new Fork();
    }

    ExecutorService executor = Executors.newCachedThreadPool();

    for ( int i = 0; i < NR_PHILOSOPHERS; i++ ) {
      Philosopher ph = new Philosopher( forks[i], forks[ (i+1) % NR_PHILOSOPHERS ], i );
      executor.execute(ph);
    }
    executor.shutdown();
  }
}
```

Running the Dining Philosophers app with deadlock detection

- Using `ThreadMXBean`
- demo

Multithreading with JavaFX

Nonresponsiveness



Active class

```
public class Task implements Runnable {
    ResponsiveGUI gui;

    public Task( ResponsiveGUI gui ) {
        this.gui = gui;
    }

    public void run(){
        gui.runTask();
    }
}
```

```
public void runTask() {
    for ( int i = 1; i <= 10; i++ ) {
        String status = "Processing " + i + " of " + 10;
        statusLbl.setText( status );
        System.out.println( status );
        takeABreak( 1000 );
    }
}
```

Create and start a thread

```
public void startTask() {
    Task task = new Task( this );
    Thread taskThread = new Thread( task );
    taskThread.start();
}
```

After pressing the Start button you get:

```
Exception in thread "Thread-4"
java.lang.IllegalStateException: Not on FX application
thread; currentThread = Thread-4
```

Multithreading with JavaFX

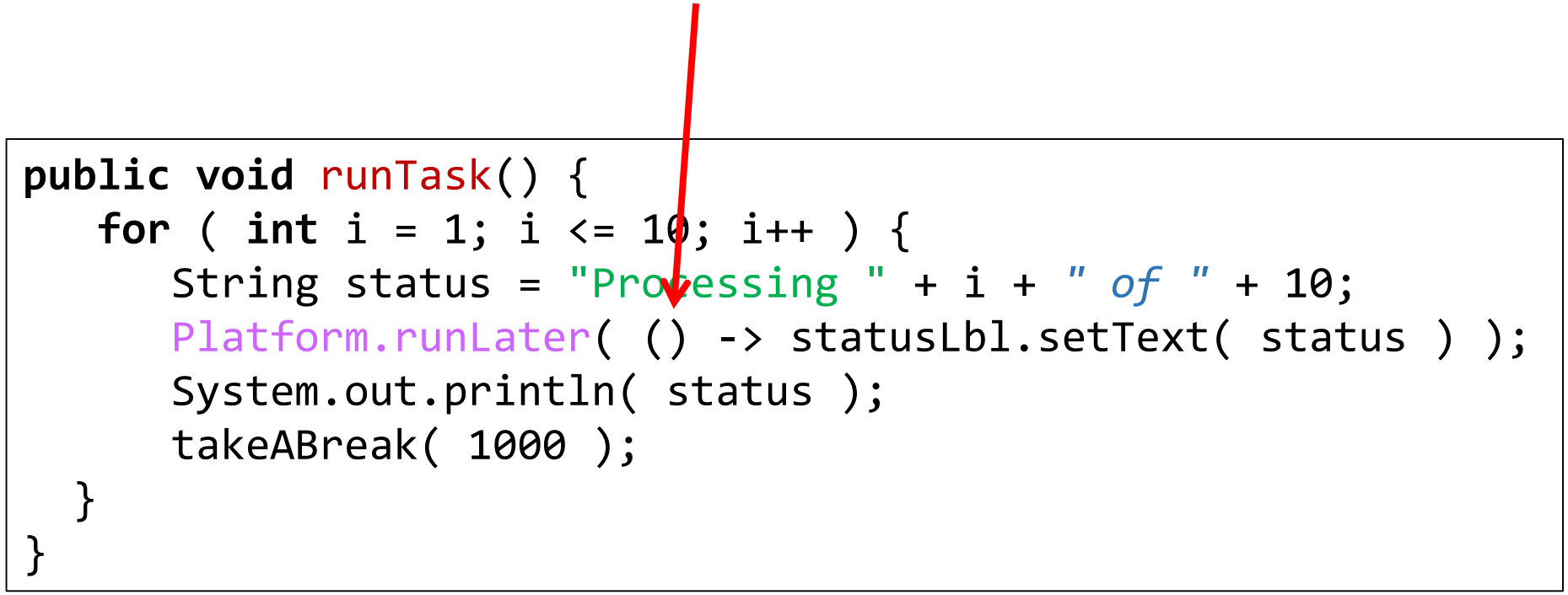
- All event handlers are executed by the JavaFX Application Thread (JAT)
- JavaFX GUI components are not thread safe.
- Thread safety in GUI applications is achieved by ensuring that JavaFX components are accessed from only the JAT.
  - Called thread confinement.
- The JavaFX runtime checks that a live scene must be accessed from the JAT.

runLater (I)

- You can dispatch a task from a user thread to the JAT by calling:

  `Platform.runLater( Runnable r )`

  – method that runs a task (Runnable r) in the JAT when appropriate.



```
public void runTask() {
   for ( int i = 1; i <= 10; i++ ) {
      String status = "Processing " + i + " of " + 10;
      Platform.runLater( () -> statusLbl.setText( status ) );
      System.out.println( status );
      takeABreak( 1000 );
   }
}
```

runLater (II)

- The gui is now responsive, however
  - pressing Exit does not stop the application!
- Terminating the JAT does not always terminate the JVM.
  - The JVM terminates if *all* running nondaemon threads terminate.

```
public void startTask() {
  Task task = new Task( this );
  Thread taskThread = new Thread( task );
  taskThread.setDaemon( true );
  taskThread.start();
}
```

Will let the JVM terminate immediately

JavaFX Concurrency Framework

- Done? Not quite.
- In general, performing a long-running task in a GUI application is not trivial.
  - decouple task-running logic from UI components
  - yield a result
  - handle errors (exceptions)
  - cancel tasks
  - show progress information

Executors

- Executors manage thread pools
  - *Executor*, a simple interface that supports launching new tasks.
  - *ExecutorService*, a subinterface of Executor, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself.
  - *ScheduledExecutorService*, a subinterface of ExecutorService, supports future and/or periodic execution of tasks.

Executors Class

- The `Executors` class provides a collection of factory methods that create thread pools which are managed using one of the three desired executor interfaces

Executor Interface

- allows submission of Runnable tasks to a thread pool via the `execute` method:

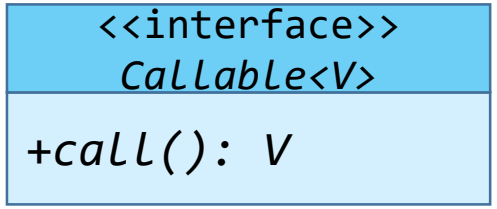
```
void execute( Runnable task )
```

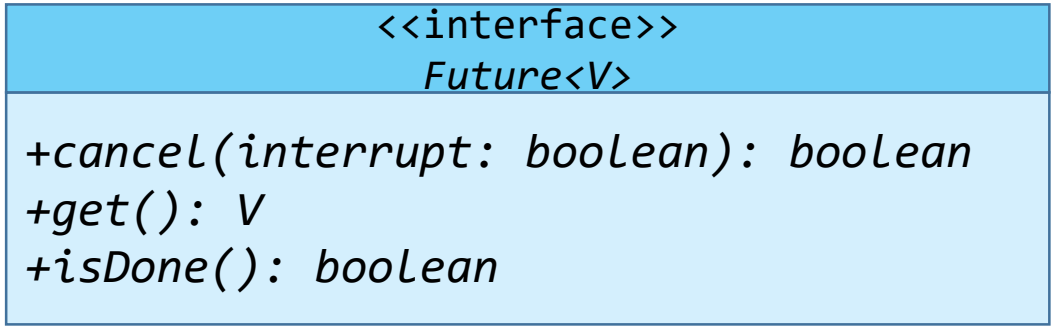
ExecutorService

- allows submission of `Runnable` or `Callable` tasks via a `submit` method

```
Future<?> submit( Runnable )
<T> Future<T> submit( Callable<T> task )
```

  - **`Callable` tasks may return a value. This value may be retrieved using the `Future` object returned by the `submit` method. It also may throw a checked exception.**
  - The `Future` object represents the *pending* result of that task.
    - You access the result using the `get` method. The thread will wait until the result is returned
    - The `Future` object also allows you to cancel the execution of the task





Example: computing Fibonacci numbers

- F(0) = F(1) = 1, F(N) = F(N-1) + F(N-2)
- Task:

```
public class FibonacciTask implements Callable<Long> {
   private int myN;

   public FibonacciTask( int n ) {
      this.myN = n;
   }

   @Override
   public Long call() throws Exception {
      return fibRecursive( myN );
   }

   private static long fibRecursive( int n ) {
      if ( n < 2 ) {
         return 1;
      } else {
         return fibRecursive( n - 1 ) + fibRecursive( n - 2 );
      }
   }
}
```

Example: computing Fibonacci numbers (2)

- Executing threads

```
public class FibonacciDemo {
 public static void main( String[] args ) {
   final int N = 44;

   ExecutorService pool = Executors.newFixedThreadPool( 2 );
   Callable<Long> c1 = new FibonacciTask( N - 1 );
   Callable<Long> c2 = new FibonacciTask( N - 2 );

   Future<Long> f1 = pool.submit( c1 );
   Future<Long> f2 = pool.submit( c2 );
   pool.shutdown();

   long fibN = 0;
   try {
      fibN = f1.get() + f2.get();
   } catch (Exception e) {
      e.printStackTrace();
   }
   System.out.println("Fibonacci number #" + N + " is " + fibN);
 }
}
```

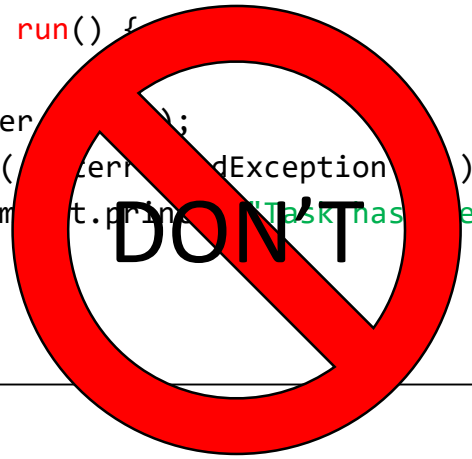
interrupt waiting/sleeping tasks

- Sometimes a program has already finished but some of the threads are still waiting/sleeping
- You can interrupt these tasks using the Thread method `interrupt`.

```
public static void main( String[] args ) {
    Task task = new Task();
    Thread taskThread = new Thread ( task );
    taskThread.start();
    taskThread.interrupt();
}
```

interrupt waiting tasks (II)

```
public class Task implements Runnable {
   private Lock myLock = new ReentrantLock();
   private Condition forever = myLock.newCondition();

   public void run() {
      try {
         forever.await();
      } catch (InterruptedException ex) {
         System.out.println("Task has been interrupted.");
      }
}
```




```
public class Task implements Runnable {
   private Lock myLock = new ReentrantLock();
   private Condition forever = myLock.newCondition();

   public void run() {
      myLock.lock();
      try {
         forever.await();
      } catch (InterruptedException ex) {
         System.out.println("Task has been interrupted.");
      } finally {
         myLock.unlock();
      }
   }
}
```

interrupt waiting tasks (III)

- This will print

```
run:
Task has been interrupted.
BUILD SUCCESSFUL (total time: 0 seconds)
```

interrupt waiting/sleeping tasks (IV)

- `if interrupt` is called while a tread was not sleeping/waiting, the thread's *interrupt status* (a boolean attribute of the Thread class) will be set.
- the Thread method `isInterupted` returns this attribute.

```
public class InfiniteTask implements Runnable {
  @Override
  public void run() {
     while (!Thread.currentThread().isInterrupted()) {
     }
     System.out.println("InfiniteTask terminated");
  }
}
public static void main( String[] args ) {
  Thread infinite = new Thread (new InfiniteTask( ));
  infinite.start();
  infinite.interrupt();
  System.out.println( "main: done!" );
}
```

```
public static Thread currentThread()

Returns a reference to the currently executing thread object.
```

```
public static void main( String[] args ) {
   Task task = new Task();
   Thread taskThread = new Thread ( task );
   taskThread.start();
   taskThread.interrupt();
   System.out.println("isInterrupted: "
         + taskThread.isInterrupted());
}
```

```
output: isInterrupted: false
```

interrupted flag only set when not sleeping/waiting

interrupt waiting tasks (V)

- Suppose an executor has been used to create the threads.
  - You cannot interrupt a thread started with `execute` directly.

```
public static void main( String[] args ) {
   Shared shared = new Shared();
   Task task     = new Task( shared );
   ExecutorService executor = Executors.newCachedThreadPool();
   executor.execute(task);
   executor.shutdown();
   try {
      boolean tasksEnded = executor.awaitTermination( 10, TimeUnit.SECONDS );
      if ( ! tasksEnded ) {
         executor.shutdownNow();
      }
   } catch ( InterruptedException ex ) {
      System.out.println( ex );
   }
}
```

interrupt waiting tasks (VI)

- You can interrupt a thread started with `submit`.

```
public static void main( String[] args ) {
    Shared shared = new Shared();
    Task task     = new Task( shared );
    ExecutorService executor = Executors.newCachedThreadPool();
    Future<?> result = executor.submit(task);
    Thread.sleep(1);
    result.cancel( true );
    executor.shutdown();
}
```

without delaying the main thread the task will not be submitted at all!

Finally