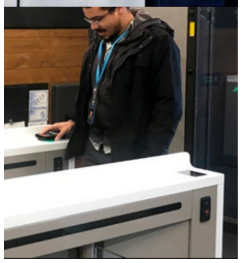
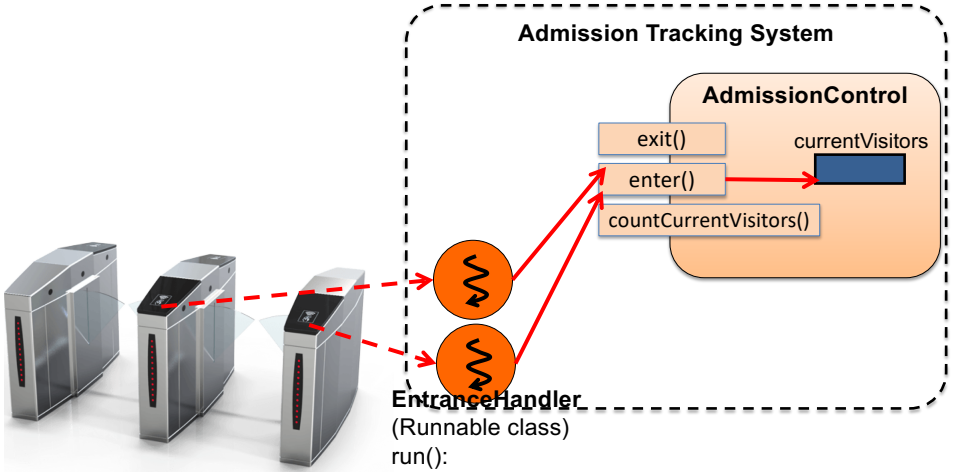
Quiz

- Electronic, real-time admission tracking and control (in a museum, for example)

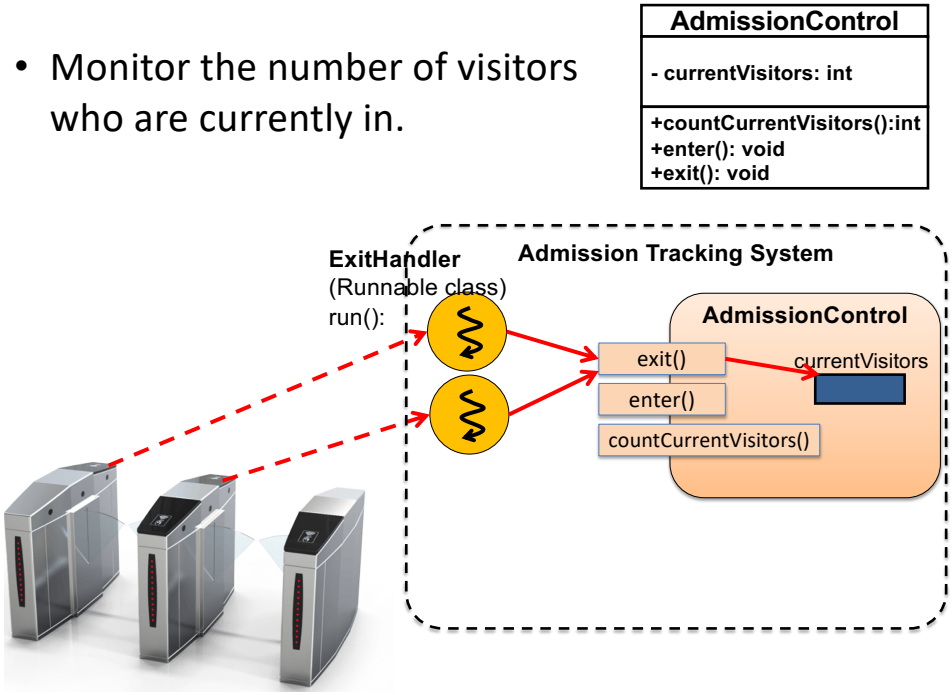



- Monitor the # of visitors who are currently in.
- Record the # of visitors per day.
- Record how long each visitor stays in.

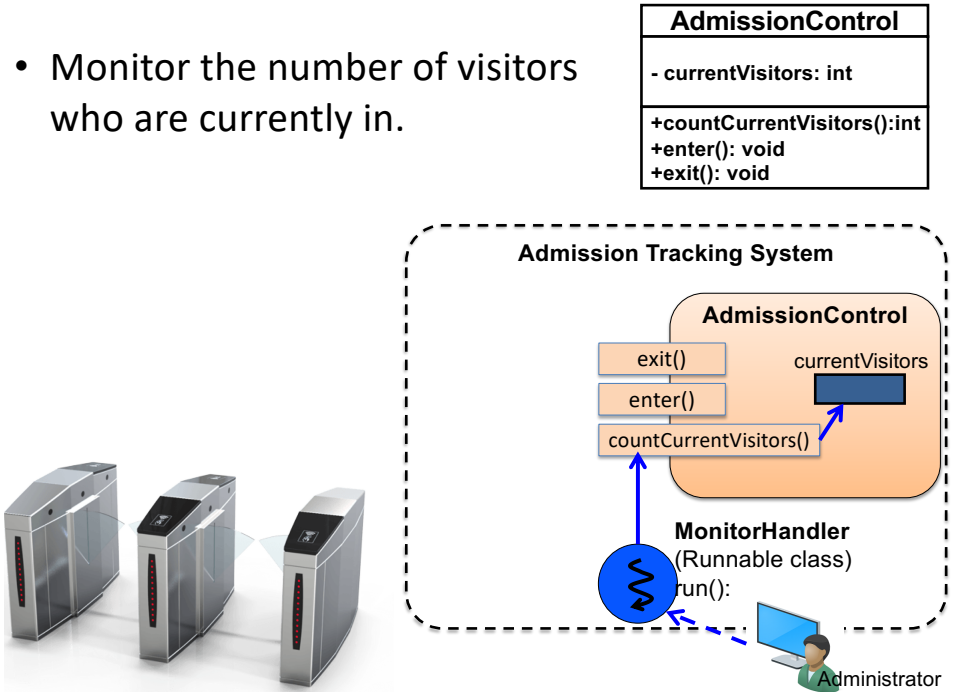
- Monitor the number of visitors who are currently in.



- Monitor the number of visitors who are currently in.

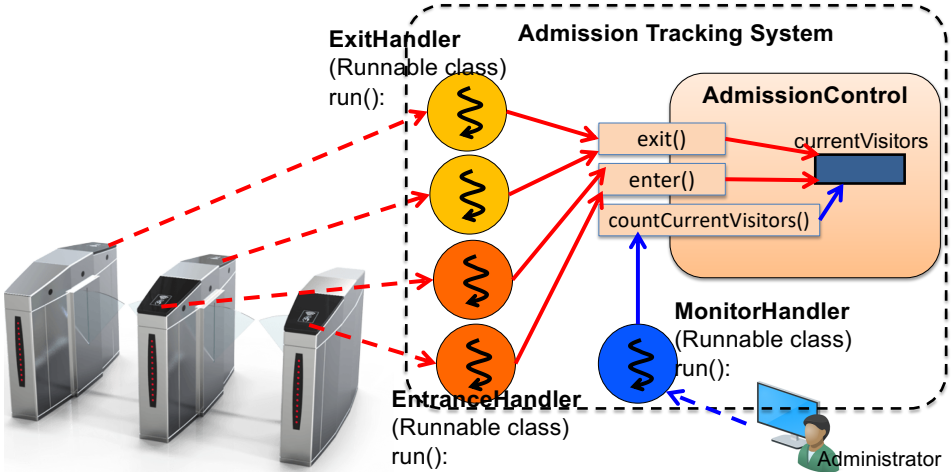


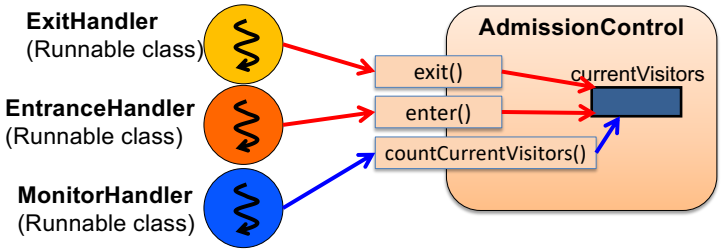
- Monitor the number of visitors who are currently in.



- Monitor the number of visitors who are currently in.

AdmissionControl
- currentVisitors: int
+countCurrentVisitors():int +enter(): void +exit(): void

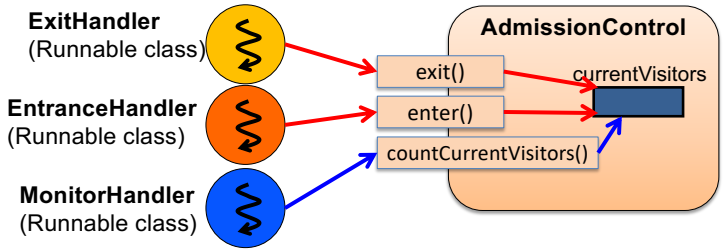





```
class AdmissionControl{
 private int currentVisitors = 0;

 public void enter(){
   currentVisitors++; }

 public void exit(){
   currentVisitors--; }

 public int countCurrentVisitors(){
   return currentVisitors; }
}
```

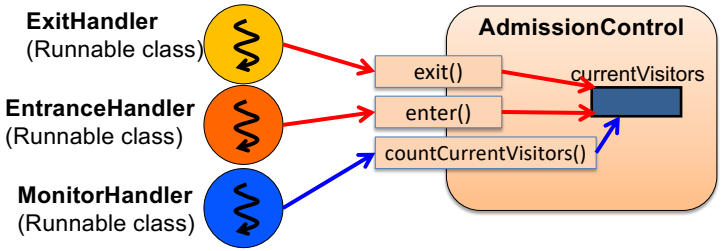
```
class EntranceHandler implements ...{
  private AdmissionControl control;
  public void run(){
    control.enter(); } }

class ExitHandler implements ...{
  private AdmissionControl control;
  public void run(){
    control.exit(); } }

class MonitorHandler implements ...{
  private AdmissionControl control;
  public void run(){
    control.countCurrentVisitors();}}
```

- **AdmissionControl is not thread-safe. Explain why and how to make it thread-safe.**




```
class AdmissionControl{
  private int currentVisitors = 0;
  private ReentrantLock lock;

  public void enter(){
    lock.lock();
    currentVisitors++;
    lock.unlock(); }

  public void exit(){
    lock.lock();
    currentVisitors--;
    lock.unlock(); }

  public int countCurrentVisitors(){
    lock.lock();
    return currentVisitors;
    lock.unlock(); } }
```

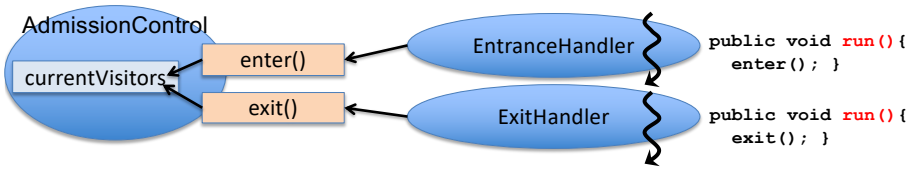
- Guard the shared variable with a lock.



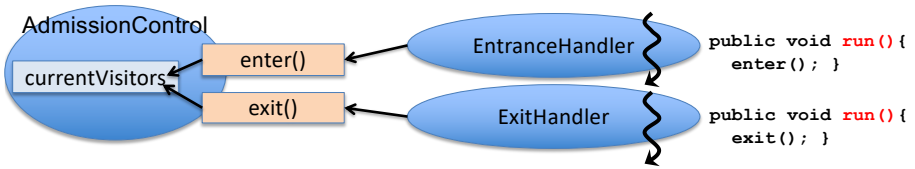

```
class AdmissionControl{
  private AtomicInteger currentVisitors;

  public void enter(){
    currentVisitors.incrementAndGet();
  }

  public void exit(){
    currentVisitors.decrementAndGet();
  }

  public int countCurrentVisitors(){
    return currentVisitors.intValue();
  }
}
```

- Use `AtomicInteger` rather than `int`.

Implement Conditional Admissions




```
• enter(){
   lock.lock();
   while(currentVisitors >= 5){
       System.out.print("Too many visitors. Please wait for a while!");
       // waiting until the # of visitors goes below 5
       Thread.sleep(1000);
   }
   currentVisitors++;
   lock.unlock(); }

• exit(double amount){
   lock.lock();
   currentVisitors--;
   lock.unlock(); }
```

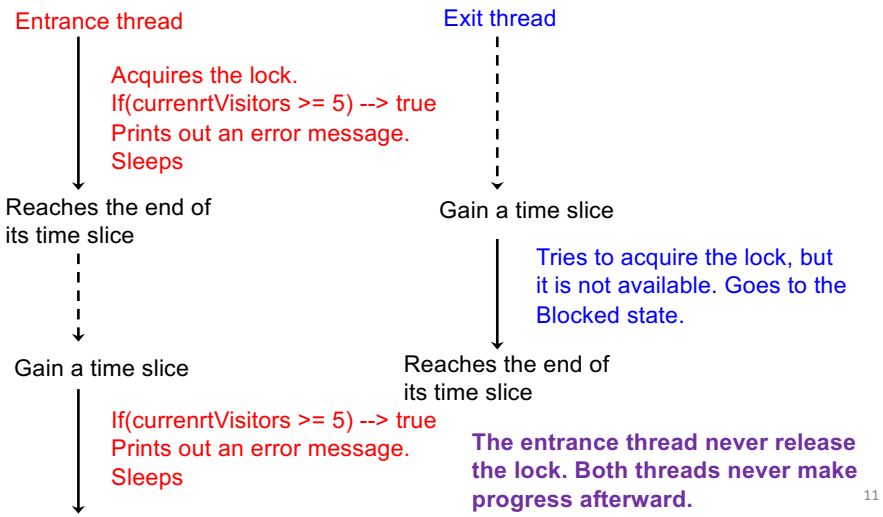



```
• enter(){
   lock.lock();
   while(currentVisitors >= 5){
       System.out.print("Too many visitors. Please wait for a while!");
       // waiting until the # of visitors goes below 5
       Thread.sleep(1000);
   }
   currentVisitors++;
   lock.unlock(); }

• exit(double amount){
   lock.lock();
   currentVisitors--;
   lock.unlock(); }
```

This code can cause a deadlock.

How Can a Deadlock Occur?

- Suppose 5 visitors are already in.



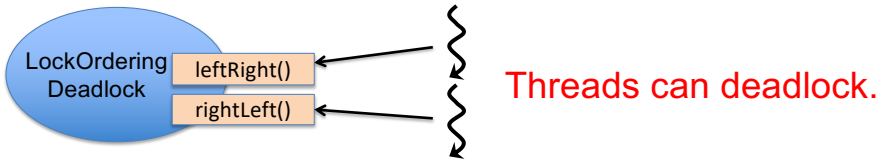
Note

- DO NOT allow a thread to conditionally stop making progress (i.e., wait until a certain condition is satisfied) with a lock held.
  - Use a condition object, so the thread can temporarily release the lock.

HW 15

- Submit a thread-safe version of **`AdmissionControl`**
  - Implement conditional admission
  - Avoid race conditions
  - Avoid deadlocks with a condition object
- Implement 2-step thread termination for the main thread to terminate "entrance" and "exit" threads.
- Deadline: April 25 (Thu) midnight

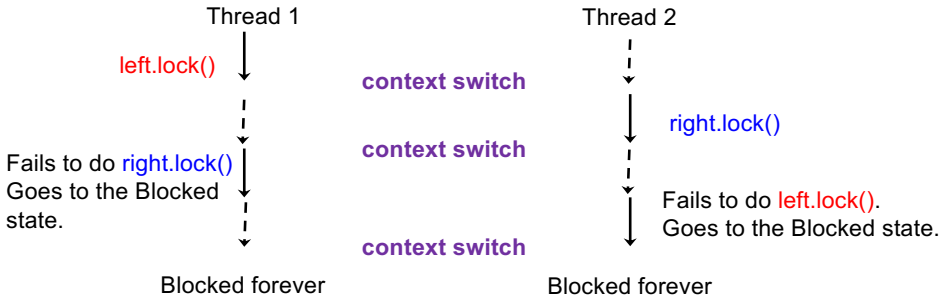
Lock-ordering Deadlocks




```
class LockOrderingDeadlock{
 private ReentrantLock left = new ReentrantLock();
 private ReentrantLock right = new ReentrantLock();

 public void leftRight(){
     left.lock();
     right.lock();
     // atomic code
     right.unlock();
     left.unlock();
 }

 public void rightLeft(){
     right.lock();
     left.lock();
     // atomic code
     left.unlock();
     right.unlock();
 }
```

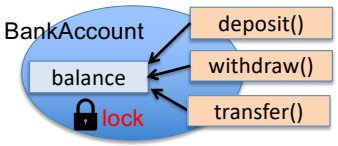
Lock-ordering Deadlocks

```
class LockOrderingDeadlock{
 private ReentrantLock left = new ReentrantLock();
 private ReentrantLock right = new ReentrantLock();

 public void leftRight(){
     left.lock();
     right.lock();      <-- A context switch can occur here.
     // atomic code
     right.unlock();
     left.unlock(); }

 public void rightLeft(){
     right.lock();
     left.lock();       <-- A context switch can occur here.
     // atomic code
     left.unlock();
     right.unlock(); }
```



Dynamic Lock-ordering Deadlocks

- Problem:
  - Threads try to acquire *the same set of locks* in *different orders*.
    - *Inconsistent* lock ordering
      - Thread 1: left → right
      - Thread 2: right → left
- To-do:
  - Have all threads acquire the locks in a *globally-fixed* order.
- **Be careful when you use multiple locks in order!**

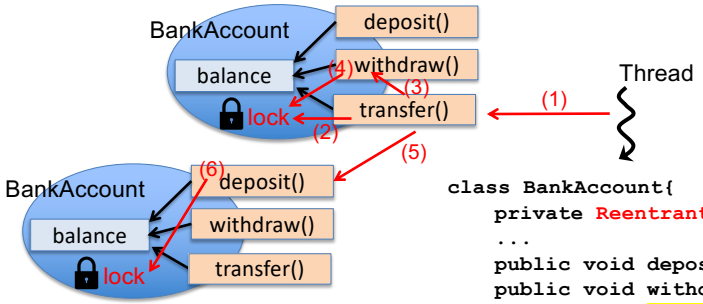



```
class BankAccount{
    private ReentrantLock lock = ...;
    ...
    public void deposit(...)
    public void withdraw(...)
    public void transfer(...) }
```

-
```
void deposit(double amount){
    lock.lock();
    balance += amount;
    lock.unlock(); }
```

```
void withdraw(double amount){
    lock.lock();
    balance -= amount;
    lock.unlock(); }
```

-
```
void void transfer(Account destination, double amount){
    lock.lock();
    if( balance < amount )
        // generate an error msg or throw an exception
    else{
        withdraw(amount);              // Nested locking. No problem.
        destination.deposit(amount);   // Acquire another lock.
    }
    lock.unlock();}
```

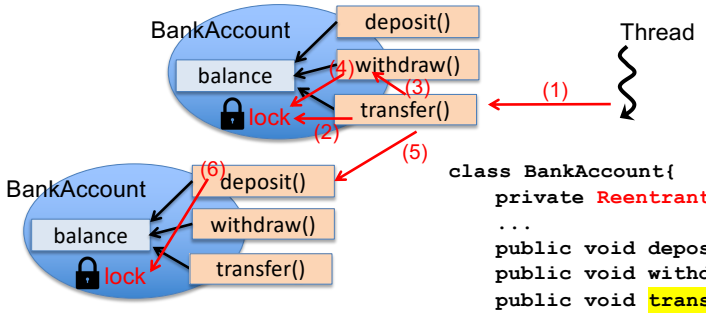



```
class BankAccount{
    private ReentrantLock lock = ...;
    ...
    public void deposit(...)
    public void withdraw(...)
    public void transfer(...) }
```

-
```
void deposit(double amount){
    lock.lock();
    balance += amount;
    lock.unlock(); }
```

```
void withdraw(double amount){
    lock.lock();
    balance -= amount;
    lock.unlock(); }
```

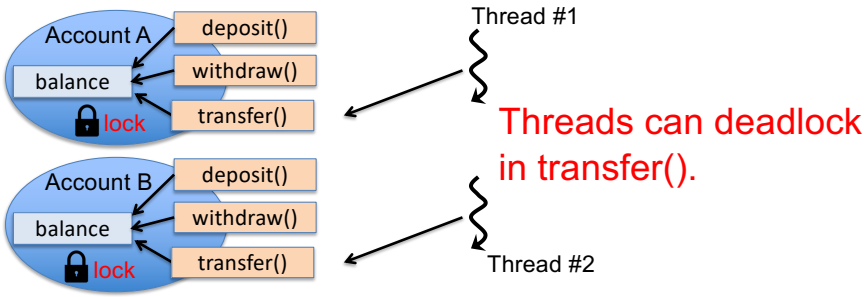
-
```
void void transfer(Account destination, double amount){
    lock.lock();
    if( balance < amount )
        // generate an error msg or throw an exception
    else{
        withdraw(amount);              // Nested locking. No problem.
        destination.deposit(amount);   // Acquire another lock.
    }
    lock.unlock();}
```




```
class BankAccount{
    private ReentrantLock lock = ...;
    ...
    public void deposit(...)
    public void withdraw(...)
    public void transfer(...) }
```

-
```
void transfer(Account destination, double amount){
    lock.lock();
    if( balance < amount )
        // generate an error msg or throw an exception
    else{
        withdraw(amount);              // Nested locking. No problem.
        destination.deposit(amount);   // Acquire another lock.
    }
    lock.unlock();}
```

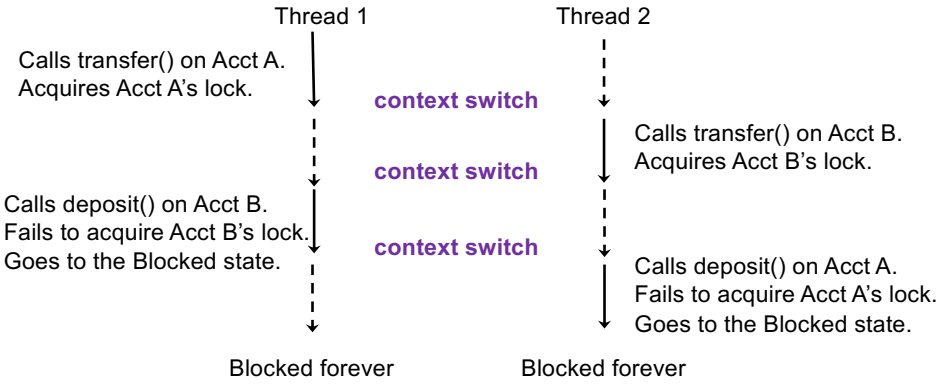
- It looks as if all threads acquire the two locks (source account's lock and destination's lock) in the same order.
- However, this code can cause a lock-ordering deadlock.



- Imagine a scenario where
  - Thread #1 transfers money from Account A to B
    - Acquires Account A's lock and B's lock.
  - Thread #2 transfers money from B to A.
    - Acquires Account B's lock and A's lock.

```
public void transfer(Account destination, double amount){
    lock.lock();
    if( balance < amount )
        // generate an error msg or...
    else{
        withdraw(amount);
        destination.deposit(amount);
    }
    lock.unlock();}
```

A context switch can occur here.



- Problem
  - Threads try to acquire *the same set of locks* in *different orders*.
    - *Inconsistent* lock ordering.
      - Thread 1: Acct A's lock → Acct B's lock
      - Thread 2: Acct B's lock → Acct A's lock
  - This can occur with bad timing although code looks OK.
    - A -> B and C -> D at the same time (No lock-ordering deadlock)
    - A -> B and A -> C at the same time (No lock-ordering deadlock)
    - A -> B and B -> A at the same time (Possible lock-ordering deadlock)
- **Be careful when you use multiple locks in order!!!**

Solutions

- Static lock
- Timed locking
- Ordered locking
- Nested tryLock()

Solution 1: Static Lock

```
private static ReentrantLock lock = new ReentrantLock();
```

```
void deposit(double amount){
    lock.lock();
    balance += amount;
    lock.unlock(); }
```

```
void withdraw(double amount){
    lock.lock();
    balance -= amount;
    lock.unlock(); }
```

```
public void transfer(Account destination, double amount){
    lock.lock();
    if( this.balance < amount )
        // generate an error msg or throw an exception
    else{
        this.withdraw(amount);          // Nested locking
        destination.deposit(amount);    // Nested locking!!!
    }
    lock.unlock();}
```

- Pros
  - Very simple
    - Uses only one lock (not two)
- Cons
  - Performance penalty
    - Deposit, withdrawal and transfers operations on different accounts are performed sequentially (not concurrently).

Solution 2: Timed Locking

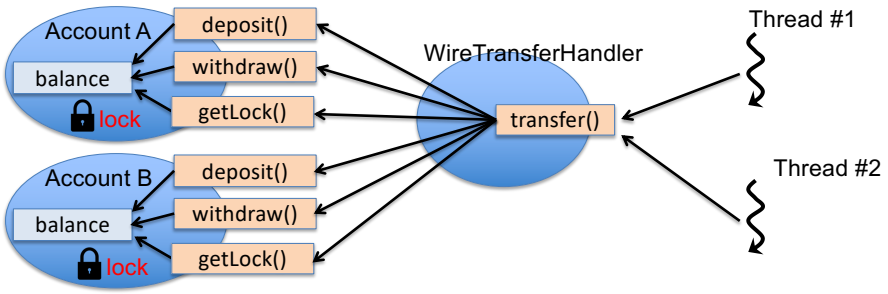
```
private ReentrantLock lock = new ReentrantLock();
```

```
public void deposit(double amount){
    if( !lock.tryLock(3, TimeUnit.SECONDS) ){
        // generate an error msg or throw an exception
    }else{
        this.balance += amount;
        lock.unlock();} }
```

```
public void transfer(Account destination, double amount){
    lock.lock();
    if( this.balance < amount )
        // generate an error msg or throw an exception
    else{
        this.withdraw(amount);      // Nested locking
        destination.deposit(amount);
    }
    lock.unlock();}  //Make sure to release this lock when
                     // an error/exception occurs.
```

- Pros
  - Simple
  - More efficient than Solution #1
    - By using a non-static lock
- Cons
  - Unprofessional
    - May need to show an unprofessional message to the user
  - Transfers and deposits might not be completed in the worst case scenario.

Solution 3: Ordered Locking



- Define a *globally-fixed* order for **`WireTransferHandler`** to acquire two locks on two accounts
- Enforce the order in **`WireTransferHandler.transfer()`**

- An example of *globally-fixed* order
  - First, acquire the lock of an account with a lower account #
  - Then, acquire the lock of an account with a higher acct #

```
public void transfer( Account source,
                      Account destination,
                      double amount){
    if( source.getAcctNum() < destination.getAcctNum() ){
        source.getLock().lock();
        destination.getLock().lock();
        if( source.getBalance() < amount )
            // generate an error msg or throw an exception
        else{
            source.withdraw(amount);        // Nested locking
            destination.deposit(amount);    // Nested locking
        }
        destination.getLock().unlock();
        source.getLock().unlock();
    }
    else if( source.getAcctNum() > destination.getAcctNum() ){
        destination.getLock().lock();
        source.getLock().lock();
        ...
        source.getLock().unlock();
        destination.getLock().unlock();
    }
```

- Pros
  - Locks are always acquired in a globally-fixed order.
  - More efficient than Solution #1
    - By using a non-static lock
  - More professional than Solution #2
    - Transfers and deposits complete for sure.
- Cons
  - Using an application-specific/dependent data (acct #)
  - Once an account is set up, its account number should not be changed.
    - If you allow dynamic changes of an account number, you need to use an extra lock in **`BankAccount`**.

Solution 3a: Ordered Locking with Instance IDs

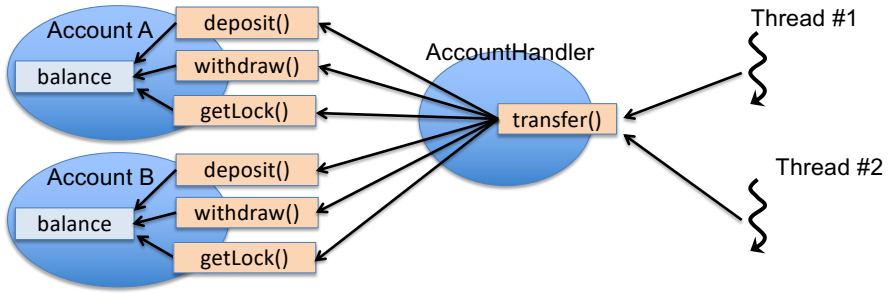
- Instance IDs
  - Unique IDs (hash code) that the local JVM assigns to individual class instances.
    - Unique and intact on the same JVM
      - 2 instances of the same class have different IDs.
      - No instances share the same ID.
      - IDs never change after they are assigned to instances.
  - Use **`System.identityHashCode(Object object)`**

- An example of *globally-fixed* order
  - First, acquire the lock of an account with a "smaller" ID
  - Then, acquire the lock of an account with a "bigger" ID

```
public void transfer( Account source,
                      Account destination,
                      double amount){
    int sourceID = System.identifyHashCode(source);
    int destID = System.identifyHashCode(destination);

    if( sourceID < destID ){
        source.getLock().lock();
        destination.getLock().lock();
        if( source.getBalance() < amount )
            // generate an error msg or throw an exception
        else{
            source.withdraw(amount);       // Nested locking
            destination.deposit(amount);   // Nested locking
        }
        destination.getLock().unlock();
        source.getLock().unlock();
    }
    if( sourceID > destID ){
        destination.getLock().lock();
        source.getLock().lock();
        ...
        source.getLock().unlock();
        destination.getLock().unlock();
    }
```

- Pros
  - Locks are always acquired in a globally-fixed order.
  - More efficient than Solution #1
    - By using a non-static lock
  - More professional than Solution #2
    - Transfers and deposits complete for sure.
  - No application-specific data (e.g., account numbers) are required to order locking.
- Cons
  - N/A

Solution 4: Nested Timed Locking



- Use nested **`tryLock()`** calls to implement an ALL-OR-NOTHING policy.
  - Acquire both of A's and B's locks, OR
  - Acquire none of them.
- Avoid a situation where a thread acquires one of the two locks and fails to acquire the other.

```
public void transfer(Account source,
                     Account destination,
                     double amount){
 Random random = new Random();

 while(true){
     if( source.getLock().tryLock() ){
         try{
             if( destination.getLock().tryLock() ){
                 try{
                     if( source.getBalance() < amount )
                         // generate an error msg/exception
                     else{
                         source.withdraw(amount);
                         destination.deposit(amount);
                     }
                     return;
                 }finally{
                     destination.getLock().unlock();
                 }
             }
         }finally{
             source.getLock().unlock();
         }
     }
     Thread.sleep(random.nextInt(1000));
 }
}
```

- If the first **`tryLock()`** fails, then sleep and try again.
- If the first **`tryLock()`** succeeds but the second one fails, unlock the first lock and sleep.

- Pros
  - More efficient than Solution #1
    - By using a non-static lock
  - More professional than Solution #2
    - Transfers and deposits complete for sure.
  - No application-specific data (e.g., account numbers) are required to order locking.
- Cons
  - Not that simple