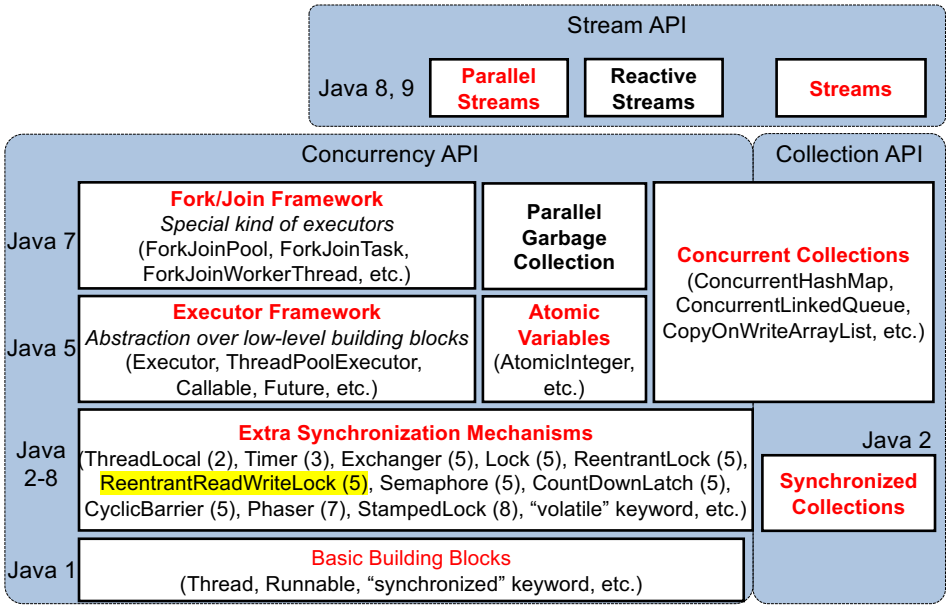
Optimistic Thread Sync (Optimistic Locking) with Read-Write Locks

Concurrency API in Java



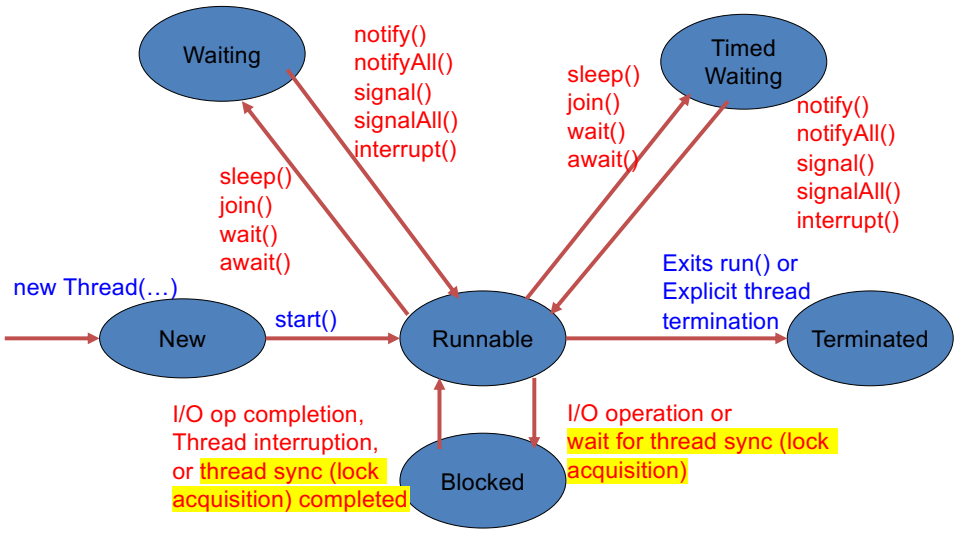
Read-Write Locks

- Regular lock (`ReentrantLock`)
  - Used to avoid race conditions by mutually excluding multiple threads that access a shared variable.
    - Allows only one of them to access the variable at a time.
- Read-Write lock
  - A slight extension to `ReentrantLock`
  - A bit more *optimistic* than `ReentrantLock` to seek performance improvement.
    - `java.util.concurrent.locks.ReentrantReadWriteLock`

Thread Sync can be Computationally Expensive

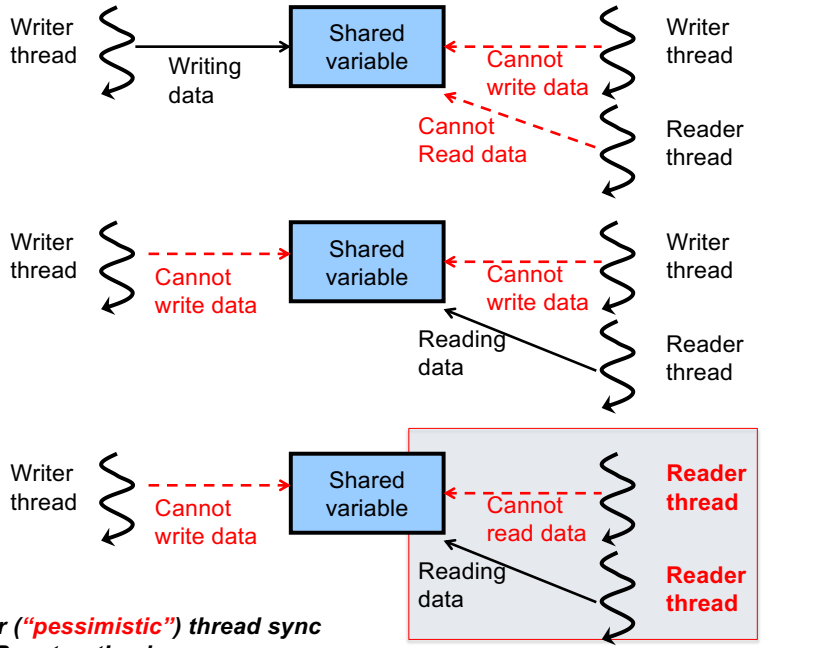
- It takes some time for each thread to acquire and release a lock.
- If a lock is not available when a thread tries to acquire it, the thread is placed to the Blocked state.
  - It does nothing while it is in the Blocked state (i.e., until it can acquire the lock).
- If you have a lot of threads that compete for the same lock, many of them may not make any progress for a long time.

States of a Thread



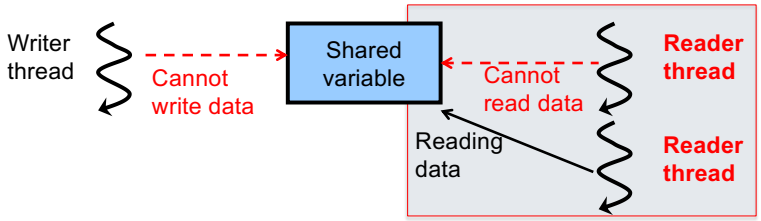
Potential Performance Improvement

- If all threads are trying to read data from a shared variable, you don't have to mutually exclude those "readers."
  - The "reader" threads will never update the variable.
  - You never have to worry about race conditions if the value of a shared variable never change.
- You can be *optimistic* NOT to mutually exclude "reader" threads.

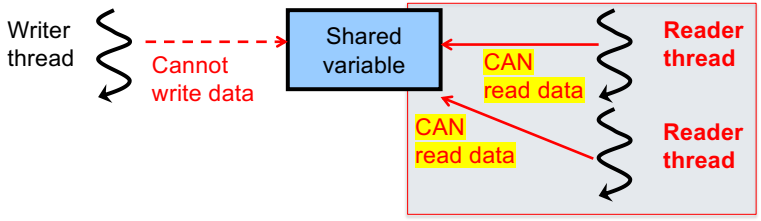


Regular ("pessimistic") thread sync with a ReentrantLock

You don't have to mutually exclude these readers.



"Pessimistic" thread sync with a ReentrantLock



"Optimistic" thread sync with a ReentrantReadWriteLock

ReentrantReadWriteLock

```
public class ReentrantReadWriteLock implements ReadWriteLock{

   public class ReentrantReadWriteLock.ReadLock implements Lock{...}

   public class ReentrantReadWriteLock.WriteLock implements Lock{...}

   public ReentrantReadWriteLock.ReadLock readLock(){...}
   public ReentrantReadWriteLock.WriteLock writeLock(){...} }
```

- Provides two locks as inner classes
  - `ReadLock` for reader threads to read data from a shared variable.
  - `WriteLock` for writer threads to write data to a shared variable.
- Provides factory methods for the two locks: `readLock()` and `writeLock()`.

- A reader can acquire a read lock even if it is already held by another reader,
  - **AS FAR AS** no writers hold a write lock.
- A writer can acquire a write lock **ONLY IF** no other writers and readers hold write/read locks.
  - Writers are mutually excluded as in pessimistic thread sync.

Can a thread acquire…? \ When another thread holds …?	ReadLock	WriteLock
ReadLock	**Yes**	No
WriteLock	No	No

This turns to be NO if you use a regular "pessimistic" lock.

When to Use Optimistic Thread Sync?

- When many reader threads run.
- When reader threads run more often than writer threads.
- When a read operation requires a long time to be completed.

An Example Optimistic Thread Sync

- ```
  int i;          // shared variable
  ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();
  ```
- For reading data from the shared variable:
  - ```
    rwLock.readLock().lock();
    try{
        System.println(i);  // atomic code
        ...
    }finally{
        rwLock.readLock().unlock(); }
    ```
- For writing data to the shared variable
  - ```
    rwLock.writeLock().lock();
    try{
        i++;  // atomic code
        ...
    }finally{
        rwLock.writeLock().unlock(); }
    ```
    ```

`ReadLock` and `WriteLock`

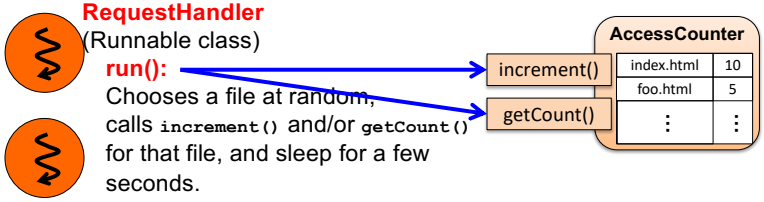
- Work similarly to `ReentrantLock`.
  - Support nested locking and thread reentrancy.
  - Support interruption via `Thread.interrupt()`.

- `WriteLock`
  - Returns a `Condition` object when `newCondition()` is called.

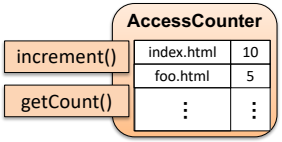
- `ReadLock`
  - Throws an `UnsupportedOperationException` when `newCondition()` is called.
    - Reader threads never need condition objects.
    - Reader threads never call `signalAll()` and `signal()`.

HW 13

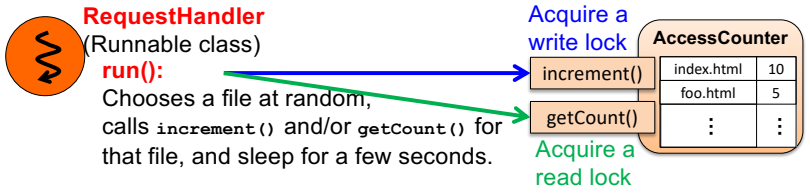
- Recall a previous HW to implement a concurrent access counter.
- `AccessCounter`
  - Maintains a map that pairs a relative file path and its access count.
    - Assume `java.util.HashMap<Path, Integer>`
  - `void increment(Path path)`
    - accepts a file path and increments its access count.
  - `int getCount(Path path)`
    - accepts a file path and returns its access count.



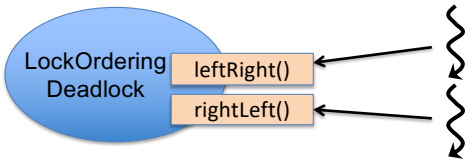
- `AccessCounter`'s `increment()` and `getCount()` need to perform thread sync.
  - `increment()`
    - lock.lock();
      if(A requested path is in AC){
        increment the path's access count. }
      else{
        add the path and the access count of 1 to AC. }
      lock.unlock();
  - `getCount()`
    - lock.lock();
      if(A requested path is in AC){
        get the path's access count and return it. }
      else{
        return 0. }
      lock.unlock();



- Replace `ReentrantLock` with `ReentrantReadWriteLock` in `AccessCounter`
  - `increment()`
    - `rwLock.writeLock().lock();`
      if(A requested path is in AC){
        increment the path's access count. } // Write
      else{
        add the path and the access count of 1 to AC. } // Write
      `rwLock.writeLock().unlock();`
  - `getCount()`
    - `rwLock.readLock().lock();`
      if(A requested path is in AC){
        get the path's access count and return it. } // Read
      else{
        return 0. }
      `rwLock.readLock().lock();`



Lock-ordering Deadlocks



Threads can deadlock.

```
class LockOrderingDeadlock{
 private ReentrantLock left = new ReentrantLock();
 private ReentrantLock right = new ReentrantLock();
 private ... ;                // Shared variables

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

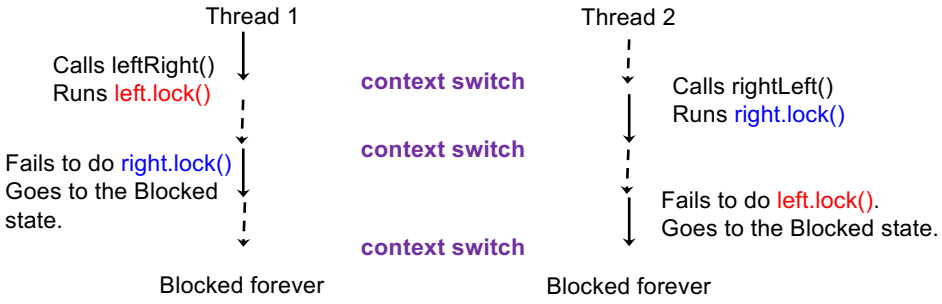
 public void rightLeft(){
     right.lock();
     left.lock();
     // atomic code to access shared variables
     left.unlock();
     right.unlock();  }}
```

Lock-ordering Deadlocks

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

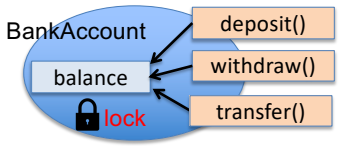
 public void leftRight(){
     left.lock();
     right.lock();   <-- A context switch can occur here.
     // atomic code
     right.unlock();
     left.unlock(); }

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



- 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!**

Dynamic Lock-ordering Deadlocks

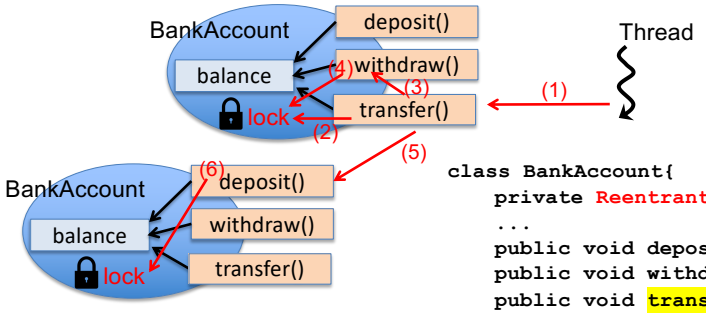



```
class BankAccount{
    private double balance = 0;
    private ReentrantLock lock = ...;
    ...
    public void deposit(...)
    public void withdraw(...)
    public void transfer(...) }
```

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

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

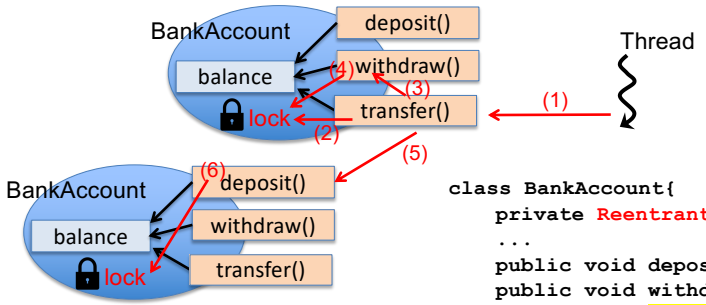
```
void void transfer(BankAccount destination, double amount){
    lock.lock();
    if( balance < amount )
      // Abort wire transfer. Generate an error msg or throw an exception
    else{
        withdraw(amount);              // Nested locking. No problem.
        destination.deposit(amount);   // 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 )
        // Abort wire transfer. Generate an error msg or throw an ...
    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.

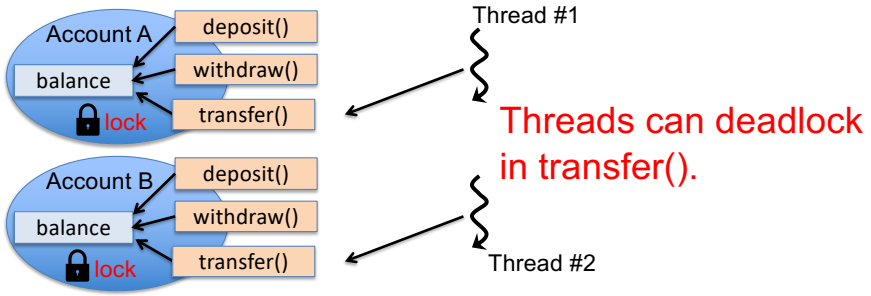



```
class BankAccount{
    private ReentrantLock lock = ...;
    ...
    public void deposit(...)
    public void withdraw(...)
    public void transfer(...) }
```

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

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

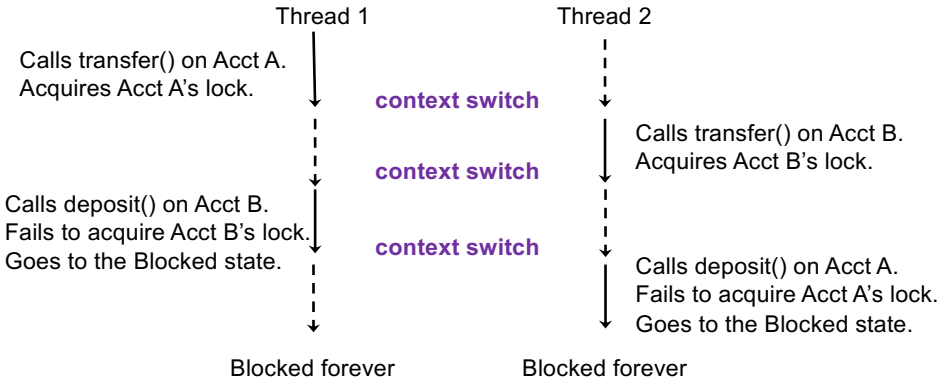
```
void void transfer(BankAccount destination, double amount){
    lock.lock();
    if( balance < amount )
      // Abort wire transfer. Generate an error msg or throw an exception
    else{
        withdraw(amount);              // Nested locking. No problem.
        destination.deposit(amount);  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



- Threads #1 and #2 can deadlock in `transfer()` if
  - #1 transfers money from Account A to B
    - Acquires Account A's lock and B's lock.
  - #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();}     // Make sure to release this lock when
                        // you abort wire transfer.
```

- Pros
  - Very simple
    - Uses only one lock (not two or more)
- Cons
  - Lack of concurrency
    - Deposit, withdrawal and transfer operations on different accounts are performed sequentially (not concurrently).
  - Performance penalty
    - All threads try to acquire a single lock. Higher chances that lock acquisition fails because the lock is not available.

Solution 2: Timed Locking

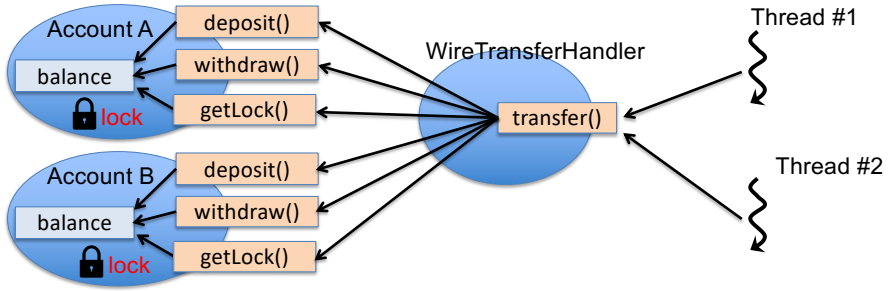
```
private ReentrantLock lock = new ReentrantLock();

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 )
        // Abort wire transfer. Generate an error msg or ...
    else{
        this.withdraw(amount);      // Nested locking
        destination.deposit(amount);
    }
    lock.unlock(); }     // Make sure to release this lock when
                         // you abort wire transfer.
```

- Pros
  - Simple
  - More efficient than Solution #1
    - By using a non-static lock
- Cons
  - Possibly 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 can complete for sure.
- Cons
  - Rely on 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`** to guard it.

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)`** OR **`object.hashCode()`**

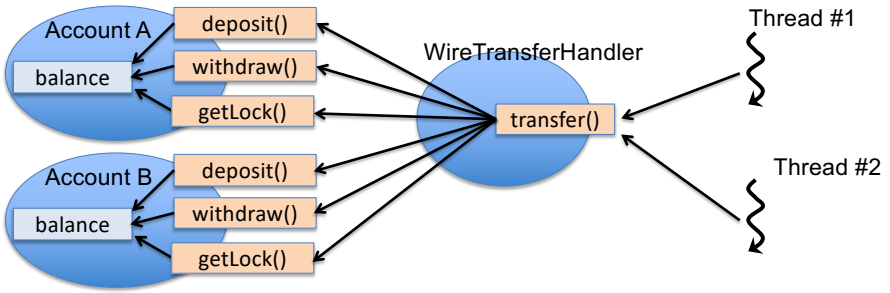
- An example of *globally-fixed* order
  - First, acquire the lock of an account with a "smaller" ID
  - Then, acquire the lock of an account with a "bigger" ID

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

    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(); }}
```

Solution 4: Nested Timed Locking

- 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



- 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();

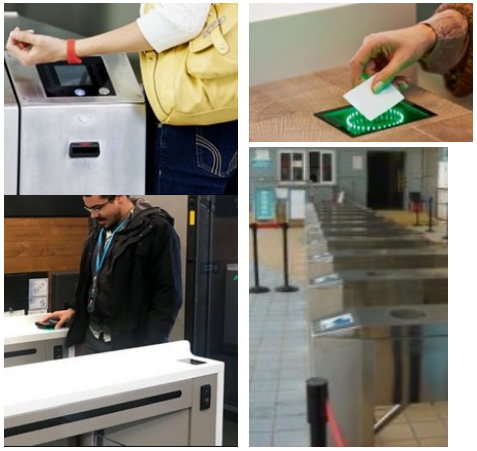
 while(true){
     if( source.getLock().tryLock() ){
         try{
             if( destination.getLock().tryLock() ){
                 try{
                     if( source.getBalance() < amount )
                         // generate an error msg/exception
                     else{
                         source.withdraw(amount);
                         destination.deposit(amount);
                     }
                     return;
                 }finally{
                     destination.getLock().unlock();
                 }
             }
         }finally{
             source.getLock().unlock();
         }
     }
     Thread.sleep( Duration.ofMillis(random.nextInt(50))); }}
```

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

- 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

Exercise: Real-time Admission Tracking

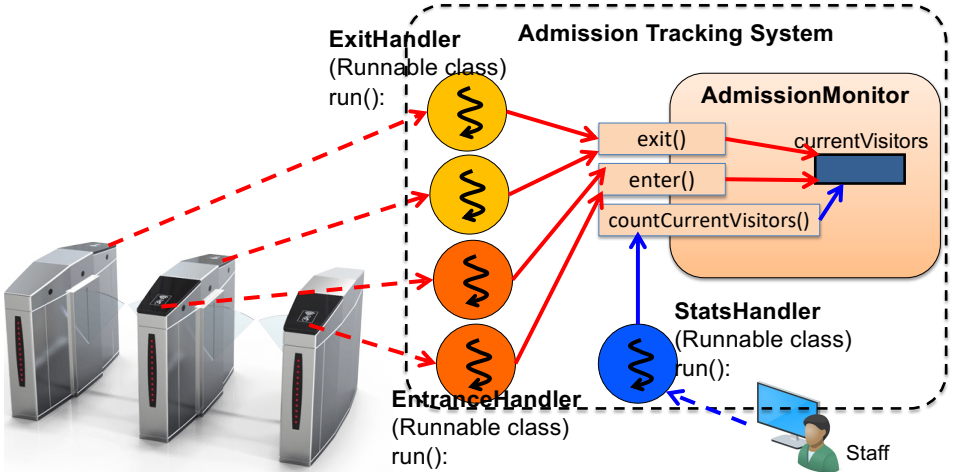
- Electronic, real-time admission tracking and control (in a museum, for example)

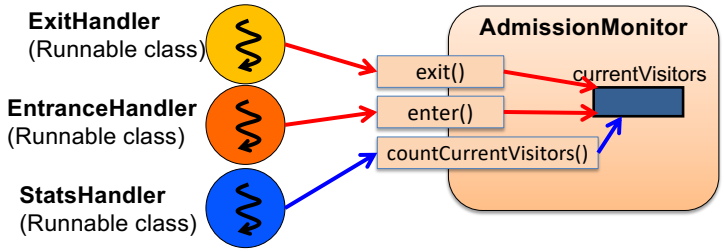


- Monitor the # of visitors who are currently in.
- Record the # of visitors per day.
- Record how long each visitor stays in.

- Monitor the number of visitors who are currently in.

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






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

 public void enter(){
   currentVisitors++; }

 public void exit(){
   currentVisitors--; }

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

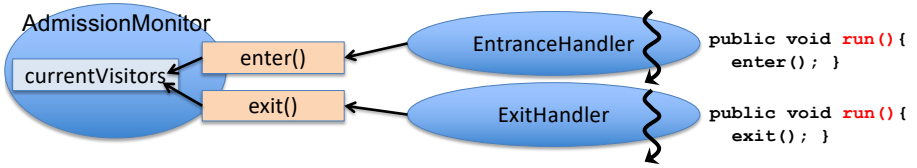
```
class EntranceHandler implements ...{
  private AdmissionMonitor monitor;
  public void run(){
    monitor.enter(); } }

class ExitHandler implements ...{
  private AdmissionMonitor monitor;
  public void run(){
    monitor.exit(); } }

class StatsHandler implements ...{
  private AdmissionMonitor monitor;
  public void run(){
    monitor.countCurrentVisitors();}}
```

AdmissionMonitor is not thread-safe.

Conditional Admission in enter()



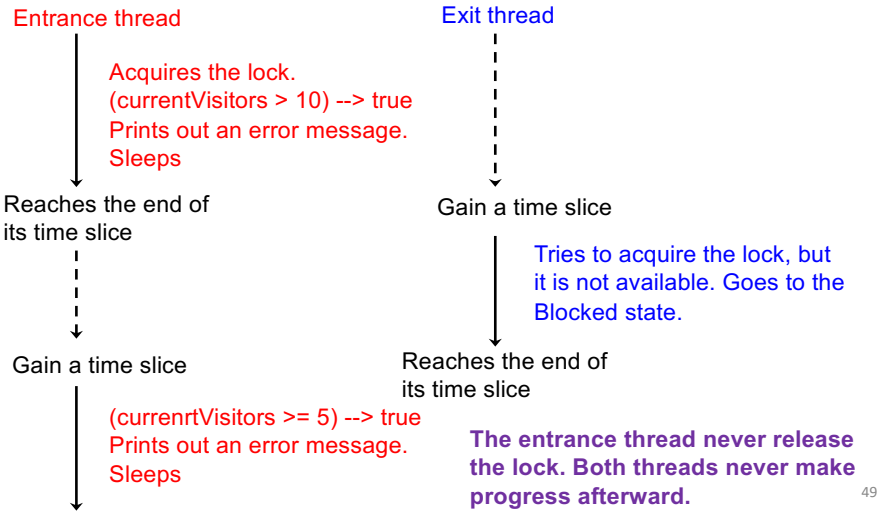

```
enter(){
 lock.lock();
 while(currentVisitors > 10){
     System.out.println("Too many visitors. Please wait for a while!");
     // waiting until the # of visitors goes below 10.
     Thread.sleep(...);
 }
 currentVisitors++;
 lock.unlock(); }
```

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

This code causes a deadlock!

How Can a Deadlock Occur?

- Suppose 15 visitors are already in.



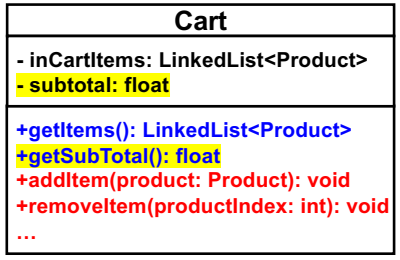
Important Note

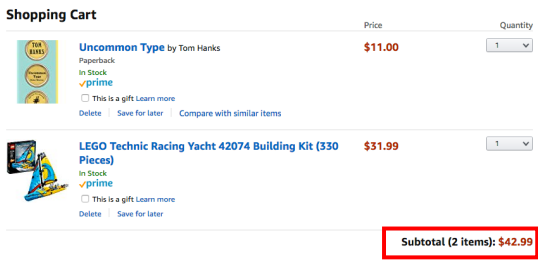
- DO NOT allow a thread to conditionally stop making progress (i.e., wait until a certain condition is satisfied) with a lock held.
  - Use a condition object, so the thread can temporarily release the lock.
  - c.f. Banking app example

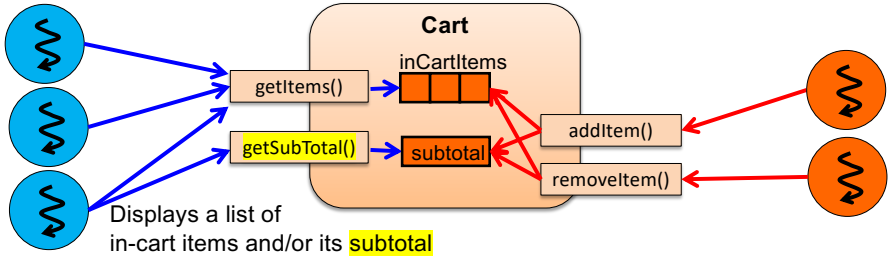
HW 13

- Submit a thread-safe version of `AdmissionMonitor`
  - Implement conditional admission
  - Avoid race conditions
    - `currentVisitors` is shared by 3 types of threads
  - Avoid deadlocks with a condition object
    - Define a `Condition` as a data field of `AdmissionMonitor`.
    - Replace `sleep()` with `await()`
    - Have `exit()` call `signalAll()`.
  - Use `ReentrantReadWriteLock` rather than `ReentrantLock`.
    - Use a write lock in `enter()` and `exit()`.
    - Use a read lock in `countCurrentVisitors()`.
- Implement 2-step thread termination for the main thread to terminate all “entrance”, “exit” and “stats” threads.

Exercise: New Feature in Shopping Carts







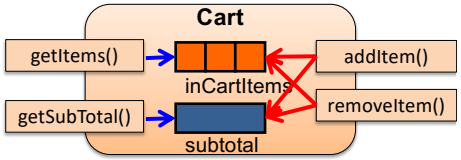
```
class Cart{
 private LinkedList<Product> inCartItems= new...
 private float subtotal;
 private ReentrantLock lock = new ...;

 public LinkedList<Product> getItems(){
   lock.lock();
   return inCartItems;
   lock.unlock();}

 public float getSubTotal(){
   return subtotal; }

 public void addItem(Product item){
   lock.lock();
   inCartItems.add(item);
   lock.unlock();
   subtotal += item.getPrice(); } }

 public void removeItem(int productIndex){
   subtotal -= inCarItems.get(productIndex).getPrice();
   lock.lock();
   inCarItems.remove(productIndex);
   lock.unlock();} }
```



- **Cart is not thread-safe. C.f. Case 1**

Using Solution 1 of Case 1

```
class Cart{
 private LinkedList<Product> inCartItems;
 private ReentrantLock lock = new ...;

 public LinkedList<Product> getItems(){
   lock.lock();
   return inCartItems;                  // READ
   lock.unlock(); }

public void addItem(Product item){
   lock.lock();
   inCartItems.add(item);               // WRITE
   lock.unlock(); }

 public void removeItem(int productIndex){
   lock.lock();
   inCarItems.remove(productIndex); // WRITE
   lock.unlock(); }

 public float getSubTotal(){                  // REVISED
   float subtotal;                            // Local variable;NOT shared among
   lock.lock();                               // multiple threads. Each thread
   for(Product item: inCarItems){   // READ // creates its own copy of it.
     subtotal += item.getPrice(); }
   lock.unlock();
   return subtotal; } }
```

- Turn a shared variable to be a local variable
  - Eliminate the data field `subtotal`.
  - Use a lock to guard `inCartItems`.

Using Solution 2 in Case 1

```
class Cart{
 private LinkedList<Product> inCartItems;
 private float subtotal;
 private ReentrantLock lock = new ...;

 public LinkedList<Product> getItems(){
   lock.lock();
   return inCartItems;                  // READ
   lock.unlock(); }

 public float getSubTotal(){
   lock.lock();
   return subtotal;                     // READ
   lock.unlock(); }

 public void addItem(Product item){
   lock.lock();
   inCartItems.add(item);               // WRITE
   subtotal += item.getPrice();         // WRITE
   lock.unlock(); }

 public void removeItem(int productIndex){
   lock.lock();
   subtotal -=
     inCarItems.get(productIndex).getPrice();   // READ, WRITE
   inCarItems.remove(productIndex);             // WRITE
   lock.unlock(); } }
```

- Use a lock to guard `inCarItems` and `subtotal`.
- It is perfectly fine to guard multiple shared variables with a single lock.