A Warning Sign (or Smell) for Deadlocks

Alien Method Call

```
class A{
  private B b;
  private ReentrantLock lock;
  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock();
  }
}
```

```
Class B{
  private A a;
  public void b1(){
    do something;
  }
}
```

- This code is deadlock-prone.
  - Calling an *alien method* with a lock held.
    - Alien methods: methods in other classes and overrideable methods in the same class
- It can cause a deadlock if an alien method (b1())…
  - Runs an infinite loop.
  - Acquires B's lock and A's lock.
    - Potential lock-ordering deadlock
  - Spawns a new thread and does a callback.

Lock-ordering Deadlock

```
class A{
  private B b;
  private ReentrantLock lock;
  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock();
  }
}
```

```
Class B{
  private A a;
  private ReentrantLock lock;
  public void b1(){
    lock.lock();
    a.a1();
    lock.unlock();
  }
}
```

- This code is deadlock-prone.
  - Calling an *alien method* with a lock held.
    - Alien methods: methods in other classes and overrideable methods in the same class
- It can cause a deadlock if an alien method (b1())…
  - Acquires B's lock and A's lock.
    - Potential lock-ordering deadlock

Lock-ordering Deadlock

```
class A{
  private B b;
  private ReentrantLock lock;
  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock();
  }
}
```

```
Class B{
  private A a;
  private ReentrantLock lock;
  public void b1(){
    lock.lock();
    a.a1();
    lock.unlock();
  }
}
```

Thread #1

Thread #2

- This code can cause a deadlock if an alien method (b1())…
  - Acquires B's lock and A's lock.
    - Potential lock-ordering deadlock
- Thread #1: acquires A's lock, B's lock and A's lock.
  - Acquires A's lock twice (nested locking): no problem
- Thread #2: acquires B's lock and A's lock.

Note

```
class A{
  private B b;
  private ReentrantLock lock;
  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock();
  }
}
```

```
Class B{
  private A a;
  public void b1(){
    do something;
  }
}
```

- If you implement A and B AND use them by yourself, be careful NOT to generate a deadlock.
- If you implement A and B as an API designer and allow others to use them, you have NO WAYS to prevent them from causing a lock-ordering deadlock.

Deadlock by a Concurrent Callback

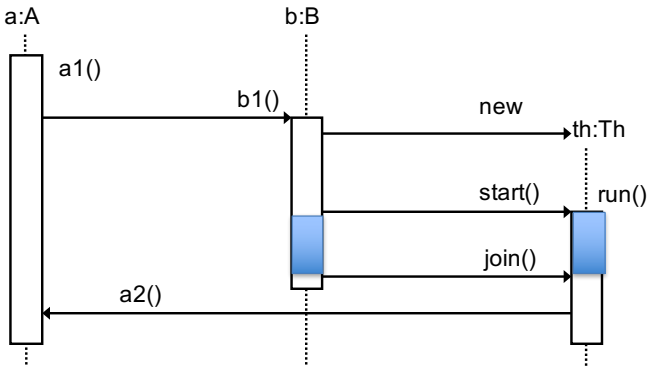
```
class A{
  private B b;
  private ReentrantLock lock;
  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock();
  }
  public void a2(){
    lock.lock();
    ……
    lock.unlock();
  }
}
```

```
Class B{
  private A a;
  public void b1(){
    ……
    Thread th = new Thread(
        new Th());
    th.start();
    ...
    th.join();
  }
}

class Th implements Runnable{
  private A a;
  public void run(){
    a.a2()
  }
}
```

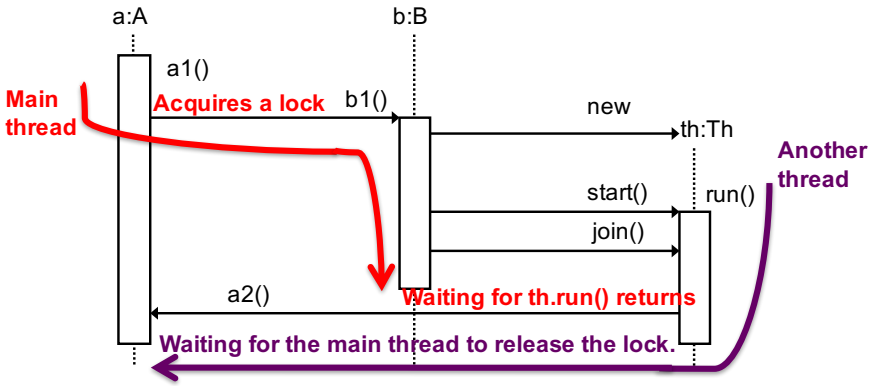
```
class A{
  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock(); }
  public void a2(){
    lock.lock();
    ……
    lock.unlock(); } }
```

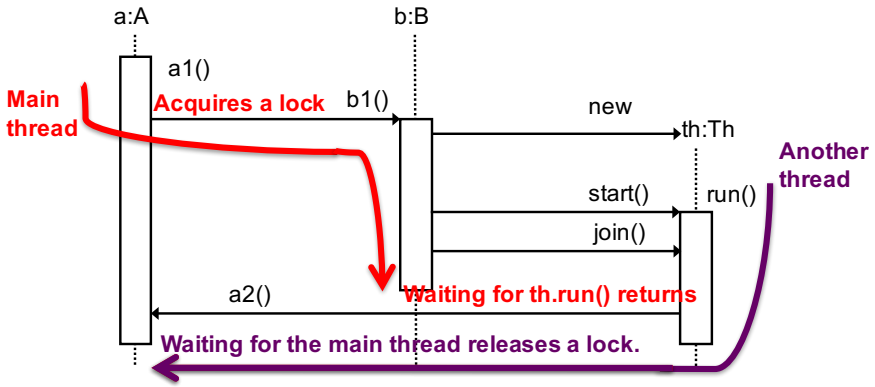
```
class B{
  public void b1(){
    Thread Th = new Thread(
        new Th());
    th.start();
    ...
    th.join(); } }
class Th implements Runnable{
  public void run(){ a.a2(); }
```




```
class A{
  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock(); }
  public void a2(){
    lock.lock();
    ……
    lock.unlock(); } }
```

```
class B{
  public void b1(){
    Thread Th = new Thread(
        new Th());
    th.start();
    ...
    th.join(); } }
class Th implements Runnable{
  public void run(){ a.a2(); }
```





- A client class holds a lock and calls an ***alien*** method.
  - Alien methods = methods in other classes and overrideable methods (neither private nor final) in the same class.
- The alien method spawns a thread that callbacks a method on the client.
  - The callback method requires a lock that the client class already holds.
- The alien method waits for a certain condition that the newly-created thread makes true.

```
class A{
  private B b;
  private ReentrantLock lock;
  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock();
  }
}
```

```
Class B{
  private A a;
  public void b1(){
    do something;
  }
}
```

- If you implement A and B AND use them by yourself, be careful NOT to generate a deadlock.
- If you implement A and B as an API designer and allow others to use them, you have NO WAYS to prevent them from making a callback in b1().

Another Example

```
class A{
  private B b;
  private ReentrantLock lock;
  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock();
  }
}
```

```
Class B{
  private A a;
  public abstract void b1(){}
}
```

- b1() is an alien method. Deadlock prone.
- If you allow others to define B's subclass and implement b1(), you have NO WAYS to prevent them from causing a deadlock.

One More Example

```
class A{
  private B b;
  private ReentrantLock lock;
  public void a1(){
    lock.lock();
    a2();
    lock.unlock();
  }
  protected abstract void a2(){}
}
```

- b1() is an alien method. Deadlock prone.
- If you allow others to define A's subclass and implement a2(), you have NO WAYS to prevent them from causing a deadlock.

If You Cannot Prevent Your Class (API) from Causing Deadlocks…

```
class A{
  private B b;
  private ReentrantLock lock;
  public void a1(){
    lock.lock();
    ...
    lock.unlock();
    b.b1(); // open call
  }
}
```

```
Class B{
  private A a;
  public void b1(){
    do something;
  }
}
```

- Open call
  - Do not call b1(), an alien method, in atomic code; move the method call outside the atomic code.

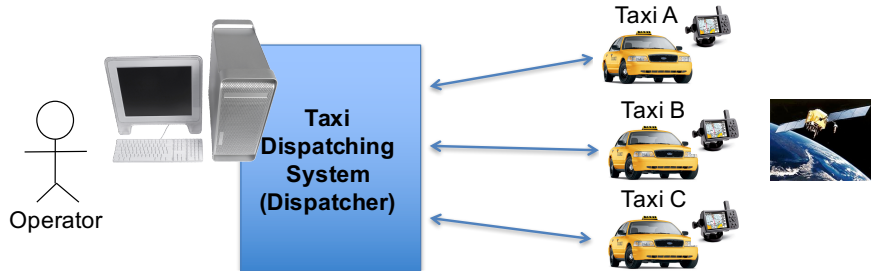
Rule of Thumb

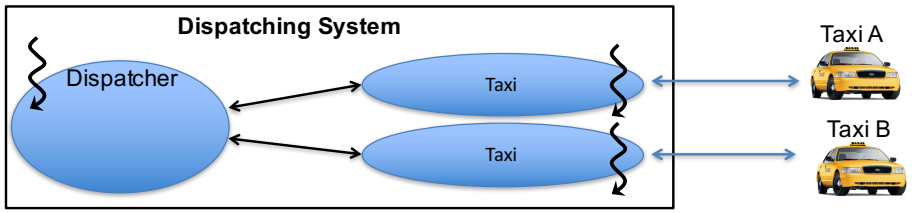
- Try NOT to call alien methods from atomic code.
- Call an alien method(s) outside atomic code.
  - Open call.
- Carefully eliminate race conditions.
- If necessary, compromise to favor being deadlock-free than being free from race conditions.

Exercise: Taxi Dispatching System

Exercise: Taxi Dispatching System

- Dispatcher
  - Periodically obtains each taxi's current location and displays it on the operator's monitor.
  - Allows the operator to set the destination location to each taxi.
- Each taxi's in-car system
  - obtains the current location and notifies the dispatcher if it has arrived at the destination.






```
class Dispatcher{
  private HashSet<Taxi> taxis;
  private Hashset<Taxi> availableTaxis;
  ……
  public void notifyAvailable(Taxi  t){
    availableTaxis.add(t);
    ……
  }

  public void displayAvailableTaxis(){
    for(Taxi t: availableTaxis)
      display.draw(t.getLocation());
    ……
  }
  …
}
```

```
class Taxi implements Runnable{
  private Point location;
  private Point dest;
  private Dispatcher dispatcher;

  public Point getLocation(){
    return location;
  }
  public setLocation(Point loc){
    location = loc;
    if(location.equals(dest))
      dispatcher.notifyAvailable(this);
  }
  public void run(){
    while(true){
      setLocation(getGPSLoc());
    }
    …… } }
```

- What variables are shared by multiple threads?
  - Taxi's location

```
Class Dispatcher{
  private HashSet<Taxi> taxis;
  private Hashset<Taxi> availableTaxis;
  ……
  public notifyAvailable(Taxi  t){
    availableTaxis.add(t);
    ……
  }

  public displayAvailableTaxis(){
    for(Taxi t: availableTaxis)
      display.draw(t.getLocation());
    ……
  }
  …
}
```

```
class Taxi implements Runnable{
  private Point location;
  private Point dest;
  private Dispatcher dispatcher;

  public Point getLocation(){
    return location;
  }
  public setLocation(Point loc){
    location = loc;
    if(location.equals(dest))
      dispatcher.notifyAvailable(this);
  }
  public void run(){
    while(true){
      setLocation(getGPSLoc());
    }
    …… } }
```

- What variables are shared by multiple threads?
  - Dispatcher's availableTaxis
    - Dispatcher could add/remove taxis to/from availableTaxis.

```
Class Dispatcher{
  private HashSet<Taxi> taxis;
  private Hashset<Taxi> availableTaxis;
  ……
  public notifyAvailable(Taxi  t){
    availableTaxis.add(t);
    ……
  }

  public displayAvailableTaxis(){
    for(Taxi t: availableTaxis)
      display.draw(t.getLocation());
    ……
  }
  …
}
```

```
class Taxi implements Runnable{
  private Point location;
  private Point dest;
  private Dispatcher dispatcher;

  public Point getLocation(){
    return location;
  }
  public setLocation(Point loc){
    location = loc;
    if(location.equals(dest))
      dispatcher.notifyAvailable(this);
  }
  public void run(){
    while(true){
      setLocation(getGPSLoc());
    }
    …… } }
```

- Guard shared variables with locks.
- Now, what about deadlocks?

```
Class Dispatcher{
  private HashSet<Taxi> taxis;
  private Hashset<Taxi> availableTaxis;
  ……
  public notifyAvailable(Taxi  t){
    lockD.lock();
    availableTaxis.add(t);
    ……
    lockD.unlock();
  }

  public displayAvailableTaxis(){
    lockD.lock();
    for(Taxi t: availableTaxis)
      display.draw(t.getLocation());
    ……
    lockD.unlock();
  }
  …}
```

```
class Taxi implements Runnable{
  private Point location;
  private Point dest;
  private Dispatcher dispatcher;

  public Point getLocation(){
    lockT.lock();
    return location;
    lockT.unlock();
  }
  public setLocation(Point loc){
    lockT.lock();
    location = loc;
    if(location.equals(dest))
      dispatcher.notifyAvailable(this);
    lockT.unlock();
  }
  public void run(){
    while(true){
      setLocation(getGPSLoc());
    }
    …… }}
```

- Now, what about deadlocks?
  - Trace threads of control and understand the order of lock acquisitions.

```
Class Dispatcher{
  private HashSet<Taxi> taxis;
  private Hashset<Taxi> availableTaxis;
  ……
  public notifyAvailable(Taxi  t){
    lockD.lock();
    availableTaxis.add(t);
    ……
    lockD.unlock();
  }

  public displayAvailableTaxis(){
    lockD.lock();
    for(Taxi t: availableTaxis)
      display.draw(t.getLocation());
    ……
    lockD.unlock();
  }
  …}
```

lockD→lockT

```
class Taxi implements Runnable{
  private Point location;
  private Point dest;
  private Dispatcher dispatcher;

  public Point getLocation(){
    lockT.lock();
    return location;
    lockT.unlock();
  }
  public setLocation(Point loc){
    lockT.lock();
    location = loc;
    if(location.equals(dest))
      dispatcher.notifyAvailable(this);
    lockT.unlock();
  }
  public void run(){
    while(true){
      setLocation(getGPSLoc());
    }
    …… }}
```

- Now, what about deadlocks?
  - Trace threads of control and understand the order of lock acquisitions.

```
Class Dispatcher{
  private HashSet<Taxi> taxis;
  private Hashset<Taxi> availableTaxis;
  ……
  public notifyAvailable(Taxi  t){
    lockD.lock();
    availableTaxis.add(t);
    ……
    lockD.unlock();
  }

  public displayAvailableTaxis(){
    lockD.lock();
    for(Taxi t: availableTaxis)
      display.draw(t.getLocation());
    ……
    lockD.unlock();
  }
  …}
```

lockD→lockT

lockT→lockD

```
class Taxi implements Runnable{
  private Point location;
  private Point dest;
  private Dispatcher dispatcher;

  public Point getLocation(){
    lockT.lock();
    return location;
    lockT.unlock();
  }
  public setLocation(Point loc){
    lockT.lock();
    location = loc;
    if(location.equals(dest))
      dispatcher.notifyAvailable(this);
    lockT.unlock();
  }
  public void run(){
    while(true){
      setLocation(getGPSLoc());
    }
    …… }}
```

- Now, what about deadlocks?
  - Two threads acquire the same set of locks in order! Be careful!!!
  - Lock-ordering deadlocks can occur.

```
Class Dispatcher{
  private HashSet<Taxi> taxis;
  private Hashset<Taxi> availableTaxis;
  ……
  public notifyAvailable(Taxi  t){
    lockD.lock(); //blocked.
    availableTaxis.add(t);
    ……
    lockD.unlock();
  }
  public displayAvailableTaxis(){
    lockD.lock();
    for(Taxi t: availableTaxis)
      display.draw(t.getLocation());
    ……
    lockD.unlock();
  }
  …}
```

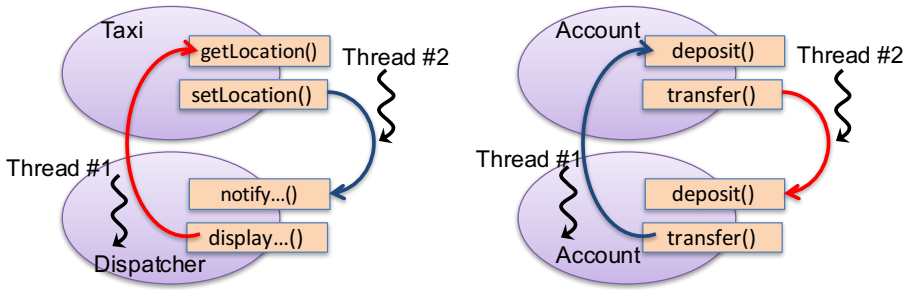
(3) (1)

```
class Taxi implements Runnable{
  private Point location;
  private Point dest;
  private Dispatcher dispatcher;

  public Point getLocation(){
    lockT.lock(); //blocked
    return location;
    lockT.unlock();
  }
  public setLocation(Point loc){
    lockT.lock();
    location = loc;
    if(location.equals(dest))
      dispatcher.notifyAvailable(this);
    lockT.unlock();
  }
  public void run(){
    while(true){
      setLocation(getGPSLoc());
    }
    …… }}
```

(4) (2)

Lock-ordering Deadlocks



Common Solutions

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

An Alternative Solution

- Open method call
  - Calling a method with no locks held.

- Now, getLocation() and notifyAvailable() are open calls.
  - No lock-ordering deadlocks occur.

```
Class Dispatcher{
  private HashSet<Taxi> taxis;
  private Hashset<Taxi> availableTaxis;
  ……
  public notifyAvailable(Taxi t){
    lockD.lock();
    availableTaxis.add(t);
    ……
    lockD.unlock();
  }

  public displayAvailableTaxis(){
    HashSet<Taxi> availableTaxisLocal;
    lockD.lock();
    availableTaxisLocal =
      new HashSet<Taxi>(availableTaxis);
    lockD.unlock();
    for(Taxi t: availableTaxisLocal)
      display.draw(t.getLocation());
      // Open call
    ……
} …}
```

```
class Taxi implements Runnable{
 private Point location;
 private Point dest;
 private Dispatcher dispatcher;

 public Point getLocation(){
   lockT.lock();
   return location;
   lockT.unlock();
 }
 public setLocation(Point loc){
   lockT.lock();
   try{
     location = loc;
     if(!location.equals(dest))
       return; // Balking
   }finally{
     lockT.unlock(); }
   dispatcher.notifyAvailable(this);
   // Open call
 }
 public void run(){...} }
```

Note 1

```
Class Dispatcher{
  private HashSet<Taxi> taxis;
  private Hashset<Taxi> availableTaxis;
  ……
  public notifyAvailable(Taxi t){
    lockD.lock();
    availableTaxis.add(t);
    ……
    lockD.unlock();
  }
```

lockT has been released when calling notifyAvailable().

A context switch can occur here. The state of "location==dest" never changes in the rest of this method.

Calling notifyAvailable() can be outside atomic code.

```
class Taxi implements Runnable{
 private Point location;
 private Point dest;
 private Dispatcher dispatcher;

 public Point getLocation(){
   lockT.lock();
   return location;
   lockT.unlock();
 }
 public setLocation(Point loc){
   lockT.lock();
   try{
     location = loc;
     if(!location.equals(dest))
       return; // Balking
   }finally{
     lockT.unlock(); }
   dispatcher.notifyAvailable(this);
   // Open call
 }
 public void run(){...} }
```

Note 2

```
Class Dispatcher{
  private HashSet<Taxi> taxis;
  private Hashset<Taxi> availableTaxis;
  ……
  public notifyAvailable(Taxi  t){
    lockD.lock();
    availableTaxis.add(t);
    ……
    lockD.unlock();
  }

  public displayAvailableTaxis(){
    HashSet<Taxi> availableTaxisLocal;
    lockD.lock();
    availableTaxisLocal =
      new HashSet<Taxi>(availableTaxis);
    lockD.unlock();
    for(Taxi t: availableTaxisLocal)
      display.draw(t.getLocation());
      // Open call
    ……
 } …}
```

```
class Taxi implements Runnable{
  private Point location;
  private Point dest;
  private Dispatcher dispatcher;

  public Point getLocation(){
    lockT.lock();
    return location;
    lockT.unlock();
  }
```

A local variable is not shared by multiple threads.

HashSet is not thread-safe.

A copy to a local variable must be in atomic code. notifyAvailable() should not be called during this copy operation.

Note 2

```
Class Dispatcher{
  private HashSet<Taxi> taxis;
  private Hashset<Taxi> availableTaxis;
  ……
  public notifyAvailable(Taxi  t){
    lockD.lock();
    availableTaxis.add(t);
    ……
    lockD.unlock();
  }

  public displayAvailableTaxis(){
    HashSet<Taxi> availableTaxisLocal;
    lockD.lock();
    availableTaxisLocal =
      new HashSet<Taxi>(availableTaxis);
    lockD.unlock();
    for(Taxi t: availableTaxisLocal)
      display.draw(t.getLocation());
      // Open call
    ……
 } …}
```

```
class Taxi implements Runnable{
  private Point location;
  private Point dest;
  private Dispatcher dispatcher;

  public Point getLocation(){
    lockT.lock();
    return location;
    lockT.unlock();
  }
```

A context switch can occur here. notifyAvailable() may be called. A race condition can occur in fact.

availableTaxisLocal may not be in synch with availableTaxis.

Note 3

```
Class Dispatcher{
  private HashSet<Taxi> taxis;
  private Hashset<Taxi> availableTaxis;
  ……
  public notifyAvailable(Taxi  t){
    lockD.lock();
    availableTaxis.add(t);
    ……
    lockD.unlock();
  }

  public displayAvailableTaxis(){
    lockD.lock();
    for(Taxi t: availableTaxis)
      display.draw(t.getLocation());
      // NOT OPEN CALL
    ……
    lockD.unlock();
 }
 …}
```

A deadlock can still occur! Need two open calls.

```
class Taxi implements Runnable{
  private Point location;
  private Point dest;
  private Dispatcher dispatcher;

  public Point getLocation(){
    lockT.lock();
    return location;
    lockT.unlock();
  }
  public setLocation(Point loc){
    lockT.lock();
    try{
      location = loc;
      if(!location.equals(dest))
        return; // Balking
    }finally{
      lockT.unlock(); }
    dispatcher.notifyAvailable(this);
    // Open call
  }
  public void run(){...} }
```

Rule of Thumb

- Try NOT to call alien methods from atomic code.
- Call alien methods outside atomic code.
  - Open call.
- Carefully eliminate race conditions.
- If necessary, compromise to favor being deadlock-free than being race condition free.

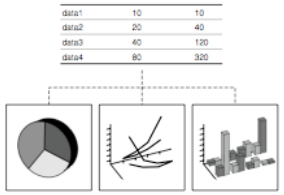
HW 22 [Optional]

- Complete thread-safe code for the taxi application
  - Use open calls

Exercise: Concurrent Observer Design Pattern

Observer Design Pattern (Recap)

- Intent
  - Event notification
    - Define a one-to-many dependency between objects so that when one object changes its state, all its dependents are notified and updated automatically
- a.k.a
  - Publish-Subscribe (pub/sub)
  - Event source - event listener
- Two key participants (classes/interfaces)
  - Observable (model, publisher or subject)
    - Propagates an event to its dependents (observers) when its state changes.
  - Observer (view and subscriber)
    - Receives events from an observable object.



Observable (publisher, event source)

registration/ subscription for an event (state change)

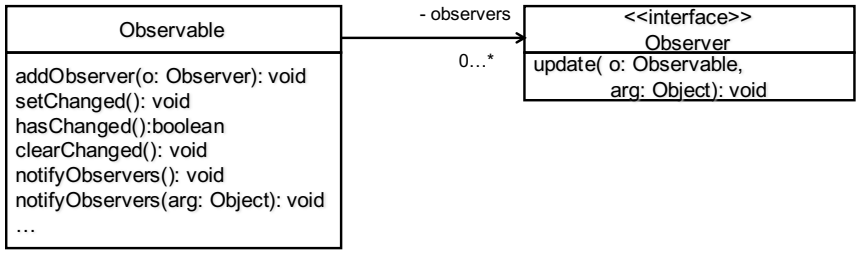
Event notification/ publication

Observers (subscribers, event listeners)

- Separate data processing from data management.
  - Data management: Observable
  - Visualization/data processing: Observers
    - e.g., Data calculation, GUI display, etc.

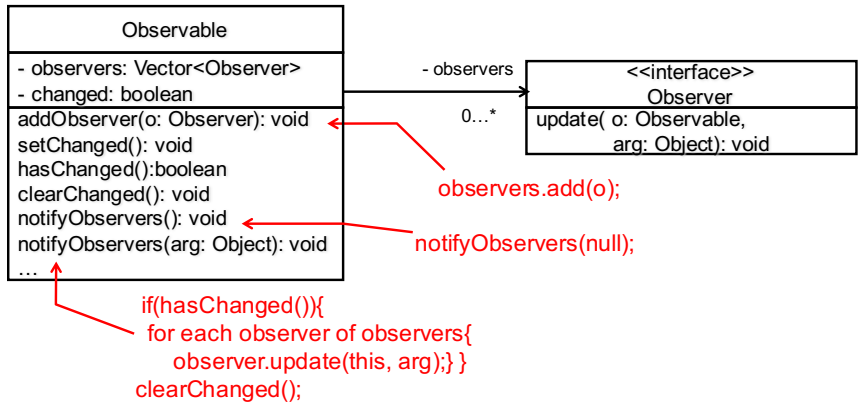
Class Structure

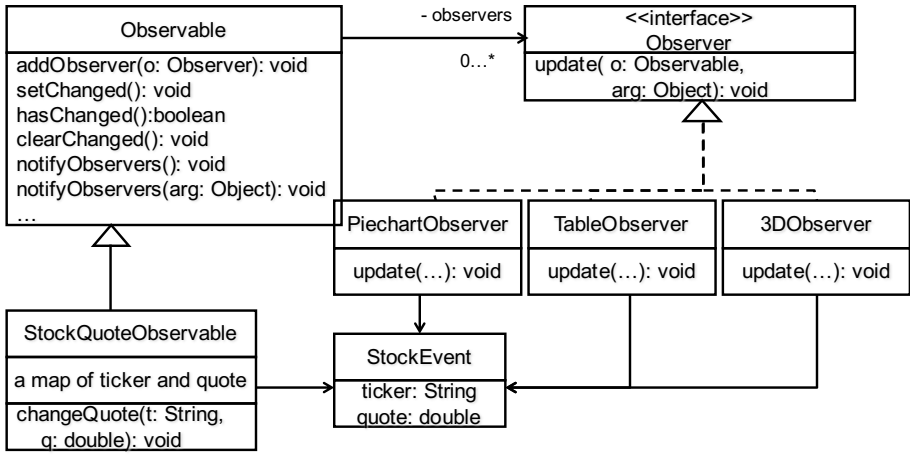
- java.util.Observable
- java.util.Observer

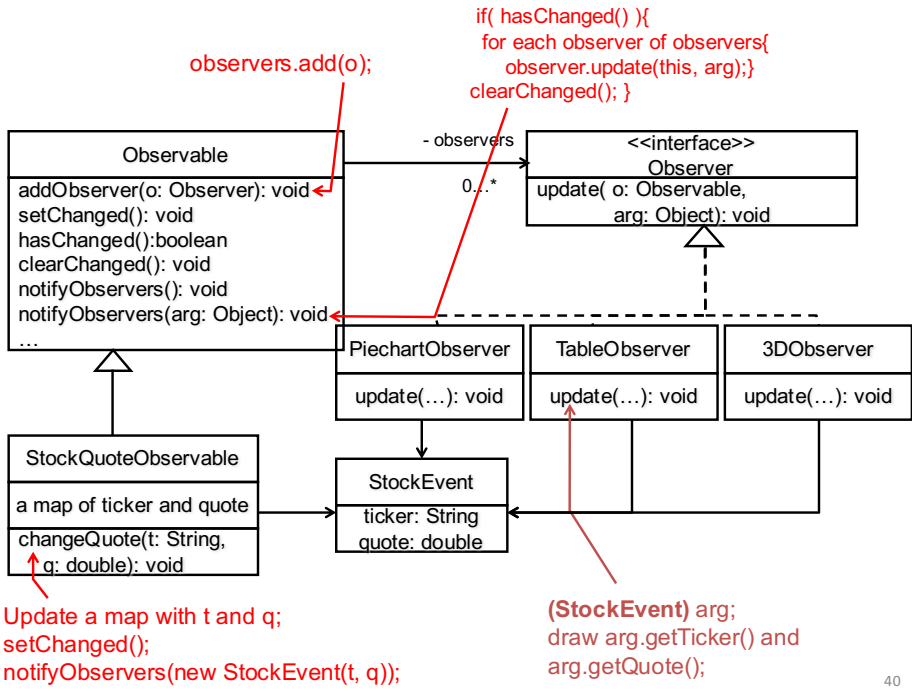


Class Structure

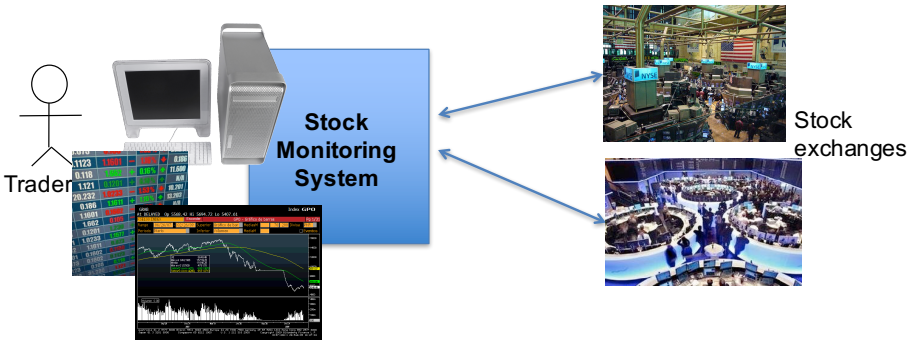
- java.util.Observable
- java.util.Observer

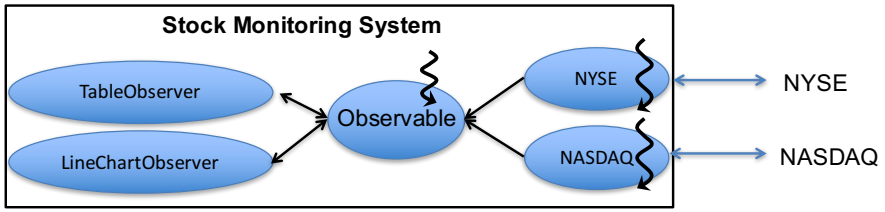


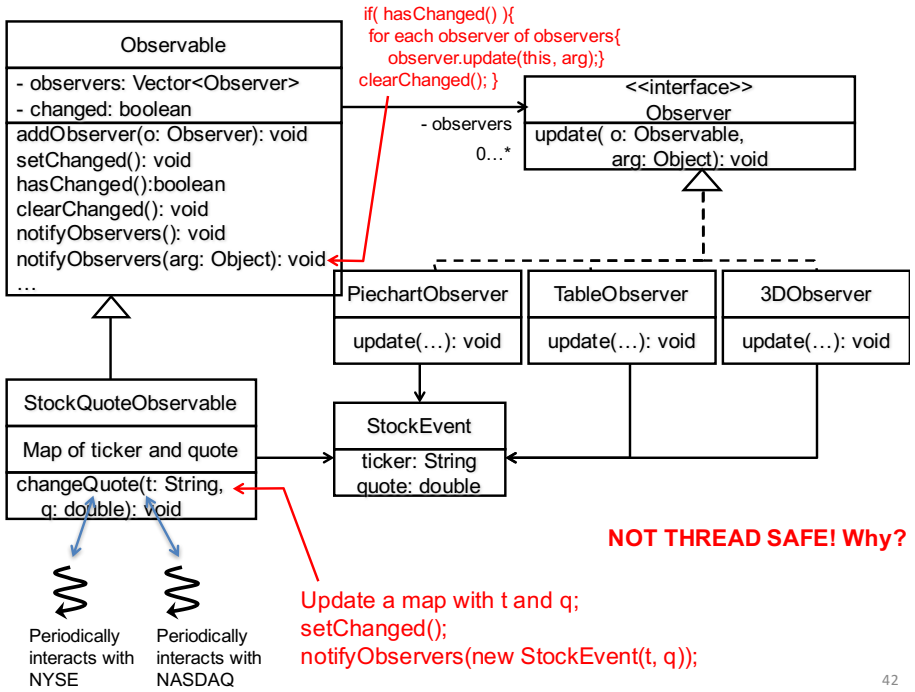


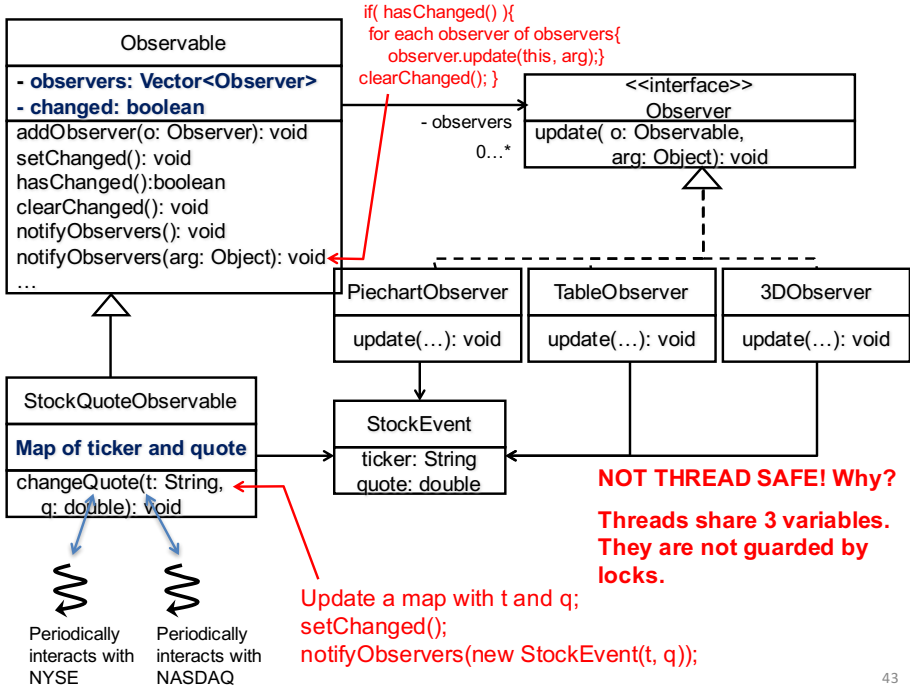


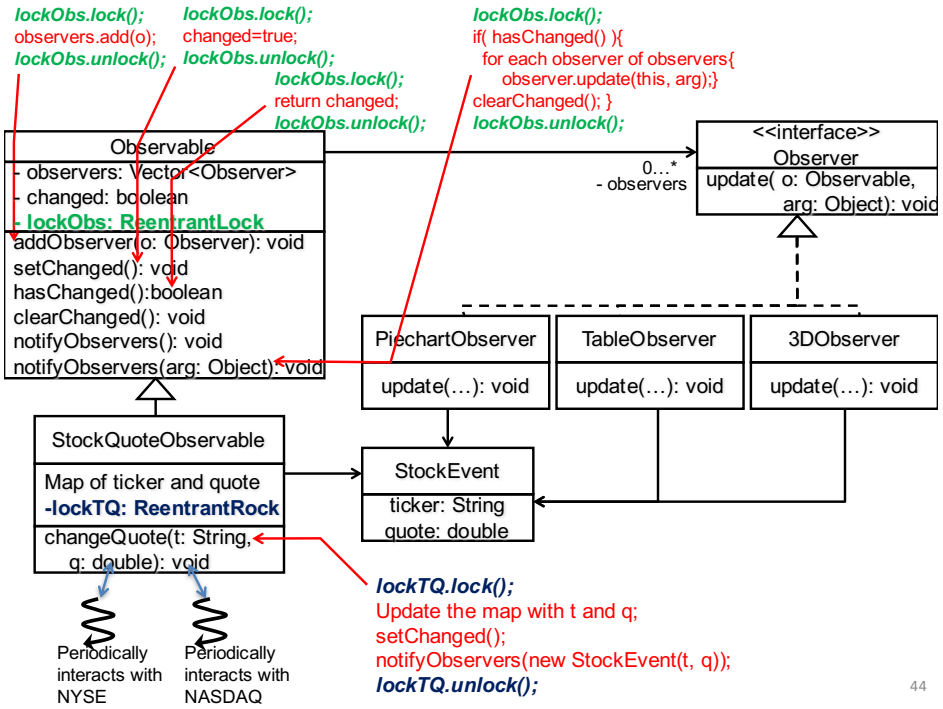
Concurrent Observer





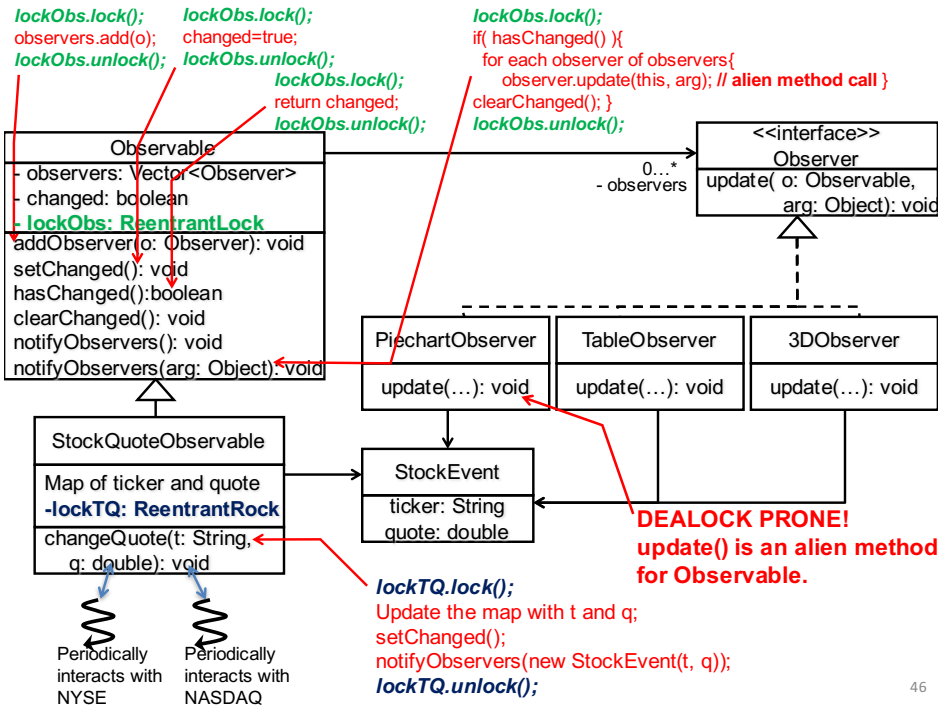






Notes

- Two threads try to acquire two locks in order in changeQuote():
  - lockTQ and lockObs
  - Be careful for potential lock-ordering deadlocks.
  - Are there any threads that try to acquire lockObs and then lockTQ?
- Observable guards two shared variables ("observers" and "changed") with a single lock.
  - No problem in terms of thread safety
    - Simple and thread-safe implementation strategy
  - Not the most efficient implementation strategy
    - Could be more efficient by guarding two shared variables with two locks.
      - One for "observers" and the other for "changed"



Potential Deadlocks due to Alien Method Calls

- update() (an alien method) may…
  - run infinite loop.
  - obtain an observer's lock and spawn a new thread that requires lockObs (i.e. the lock in Observable)
    - Lock-ordering deadlocks could occur.
  - spawn a new thread and perform a callback toward Observable.
- If you are an API designer for Observable, there are no ways for you to prevent these potential cases.
  - You need to implement Observable in a preventive way.

Open Call in Observable.notifyObservers()

```
class Observable {
  private Vector obs;              //observers
  private boolean changed = false;

  public void notifyObservers(Object arg){
    Object[] arrLocal;
    synchronized (this){           // lock.lock();
      if (!changed) return;        // balking
      arrLocal = obs.toArray();    // observers copied to arrLocal
      changed = false;
    }                              // lock.unlock();
    for (int i = arrLocal.length-1; i < =0; i--)
      ((Observer)arrLocal[i]).update(this, arg); // OPEN CALL
  }
  ……
}
```

Observable.notifyObservers()

```
class Observable {
 private Vector obs;              //observers
 private boolean changed = false;

 public void notifyObservers(Object arg){
   Object[] arrLocal;
   synchronized(this){            // lock.lock();
     if (!changed) return;        // balking
     arrLocal = obs.toArray();    // observers copied to arrLocal
     changed = false;
   }                              // lock.unlock();
   for (int i = arrLocal.length-1; i < =0; i--)
     ((Observer)arrLocal[i]).update(this, arg); // OPEN CALL
 }
 ……
}
```

A context switch can occur here. addObserver() and removeObserver() may be called. A race condition can occur.

arrLocal may not be in synch with obs.

Comments in Observable's Source Code Say…

- The worst result of any potential race-condition here is that:
  - 1) a newly-added Observer will miss a notification in progress
  - 2) a recently unregistered Observer will be wrongly notified when it doesn't care (about state-change notifications anymore).
- To look up Java API's source code…
  - Use src.zip in your JDK installation directory
  - Use online source code browsers
    - http://www.docjar.com/
    - http://grepcode.com/
    - http://javasourcecode.org/

Preventive Concurrent API Design

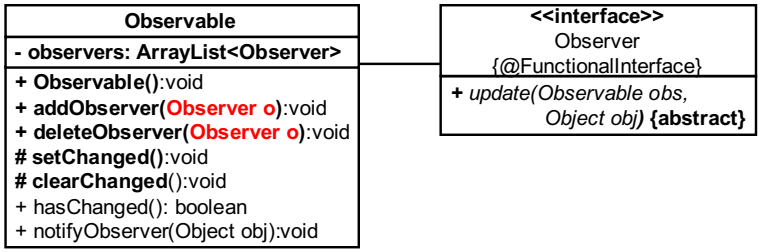
- If you implement the Observer-Observable API and its application(s) AND use both of them, be careful about potential deadlocks.
  - Deadlock by an infinite loop
  - Lock-ordering deadlock
  - Deadlock by a callback

Rule of Thumb

- Try NOT to call alien methods from atomic code.
- Call alien methods outside atomic code.
  - Open call
- Carefully eliminate race conditions.
- If necessary, compromise to favor being deadlock-free than having no race conditions.

HW 23

- Define your own Observable (class) and Observer (interface).
  - DO NOT reuse java.util.Observable and java.util.Observer.
  - Define Observer as a functional interface.
    - update() as an abstract method.
  - Implement the listed methods for Observable
    - C.f. Java API doc for expected behaviors/responsibilities for the methods
- Note that you have done this in HW 4.



- Your Observable is not thread safe.
- Revise it to be thread-safe.
  - Use open calls
  - Use ReentrantLock, not "synchronized" keyword