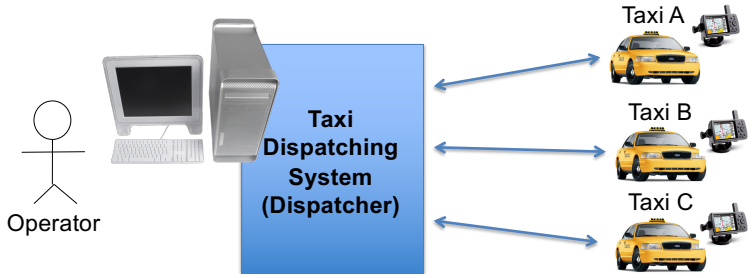
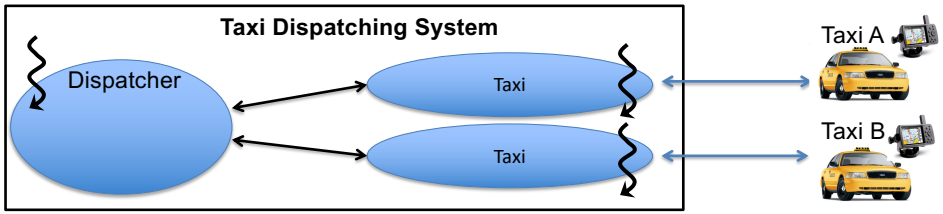
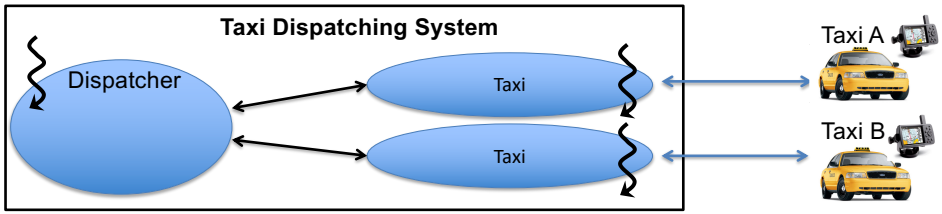
Exercise: Taxi Dispatching System

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 for picking up a customer.
- Each taxi's in-car system
  - Notifies the dispatcher if it has arrived at the destination.

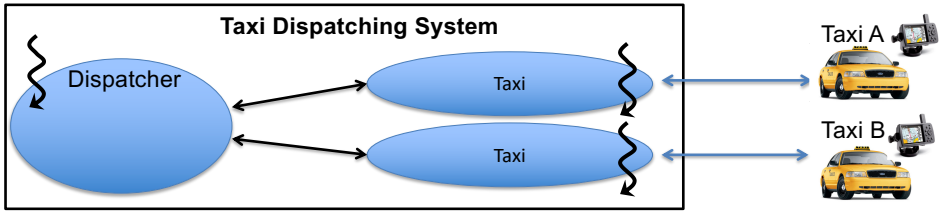






- Taxi
  - Its instances represent taxi cars on a instance-per-taxi basis.
  - Defined as a Runnable class.
    - Its run() is executed on a thread.
  - Handles taxis concurrently (not sequentially).
    - Thread-per-taxi

```
class Dispatcher{
  private HashSet<Taxi> taxis;            // All taxis
  private Hashset<Taxi> availableTaxis;   // Taxis available/ready to pick
                                          // up customers
  private Display display;
  ...
  public void notifyAvailable(Taxi t){    // Taxi calls this method to
    availableTaxis.add(t);                // tell Dispatcher that it is
    ...                                   // available/ready to pick up a
  }                                       // customer.

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




```
class Dispatcher{
  private HashSet<Taxi> taxis;
  private Hashset<Taxi> availableTaxis;
  private Display display;
  ...
  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

```
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 the 2 shared variables with 2 locks.
- Now, what about deadlocks?

```
Class Dispatcher{
  private HashSet<Taxi> taxis;
  private Hashset<Taxi> availableTaxis;
  private ReentrantLock lockD = new ...;
  ……
  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;
  private ReentrantLock lockT = …

  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 each thread of control and understand its lock acquisition(s).

```
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 each thread of control and understand its lock acquisition(s).

```
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) T thread

(1) D thread

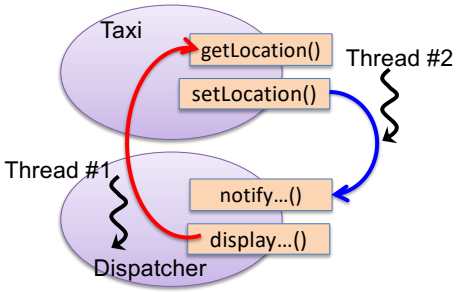
(4) D thread

(2) T thread

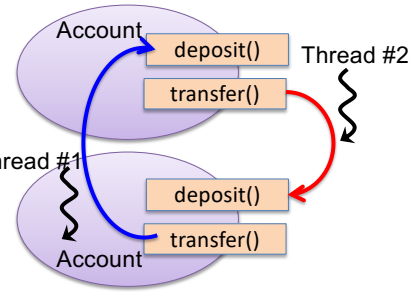
```
class Taxi implements Runnable{
  private Point location;
  private Point dest;
  private Dispatcher dispatcher;

  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());
    }
    …… }}
```

Lock-ordering Deadlocks

Taxi dispatching example



Bank account example



Common Solutions

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

An Alternative Solution

- Open method call
  - Calling an alien 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;
   }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 before calling notify…().

notifyAvailable() can be safely called 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;
   }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 never shared among 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. Race conditions can occur in fact.

availableTaxisLocal may not be in synch with availableTaxis.

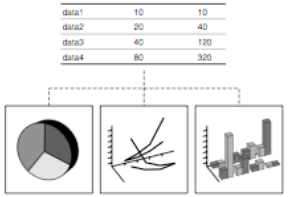
Rule of Thumb

- 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.

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)

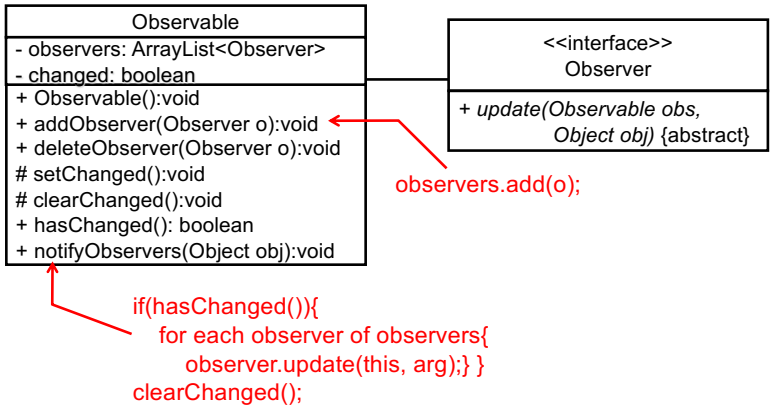
Event notification/ publication

Observers (subscribers, event listeners)

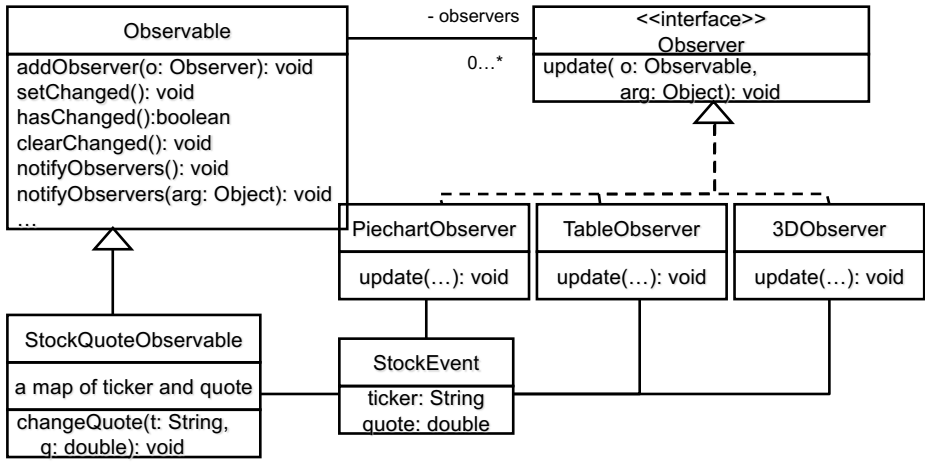
- Separate data processing from data management.
  - Data management: Observable
  - Visualization/data processing: Observers
    - e.g., Data calculation, GUI display, etc.

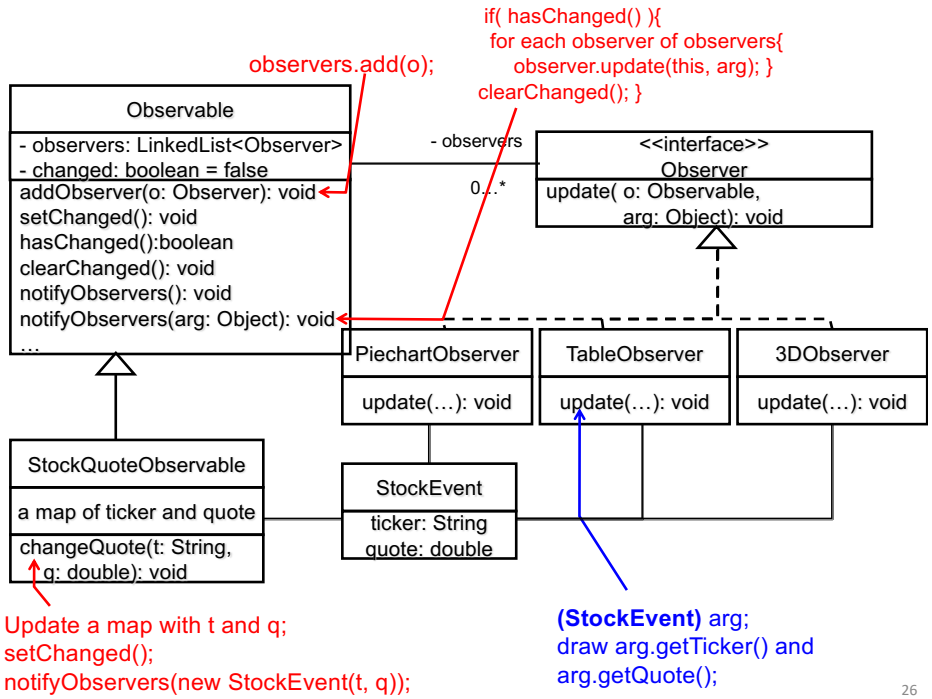
Observable and Observer

- java.util.Observable and java.util.Observer
  - c.f. CS680 and CS681's HW 1

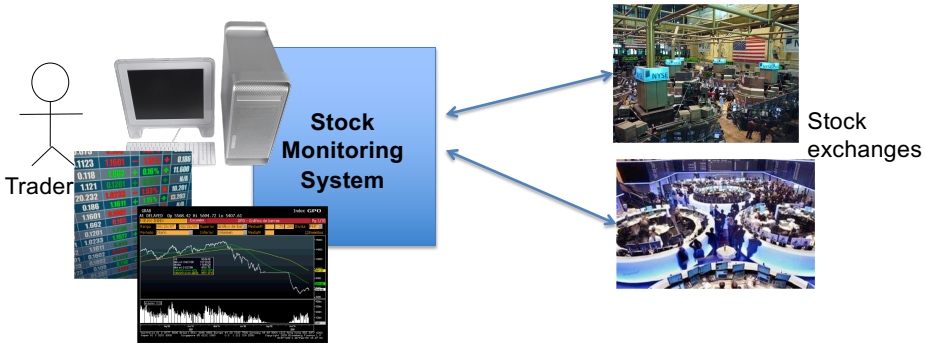


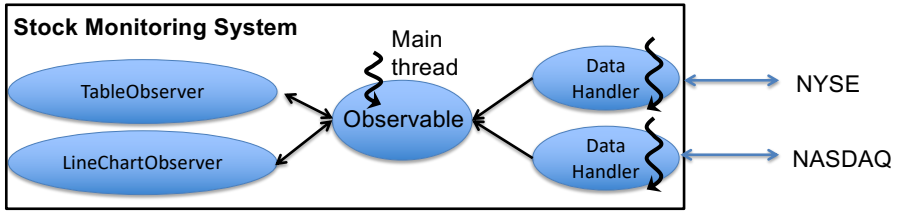
An Example of *Observer*

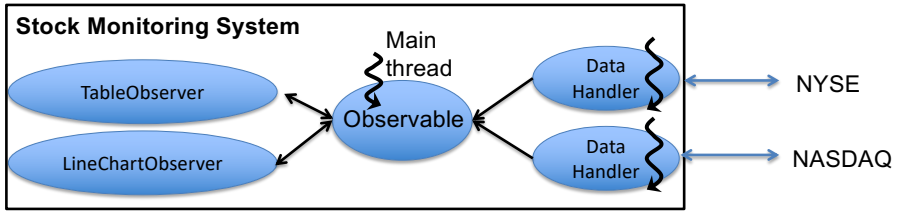




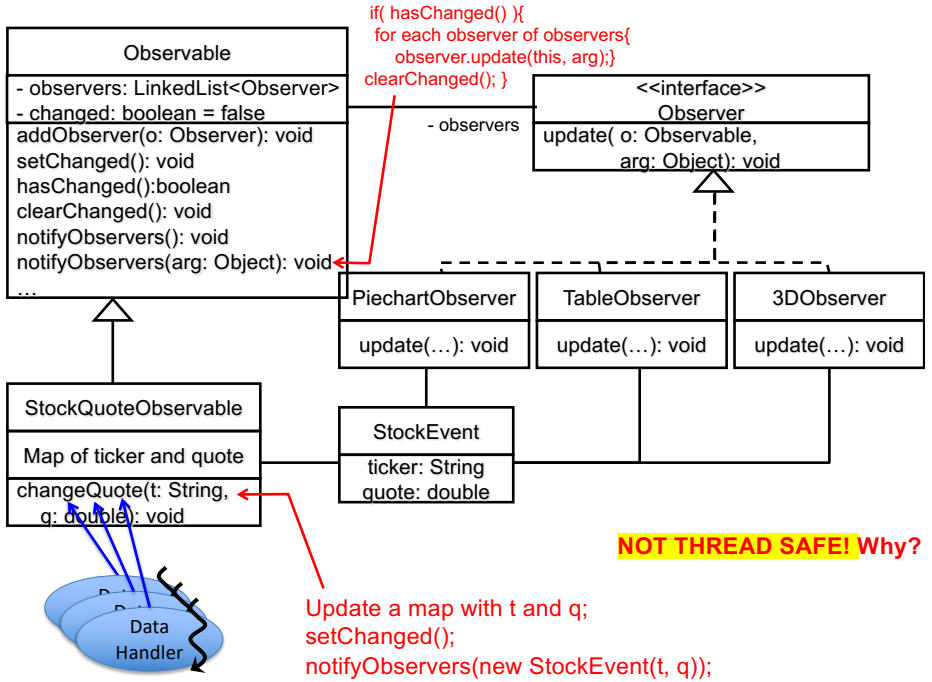
Concurrency in *Observer*

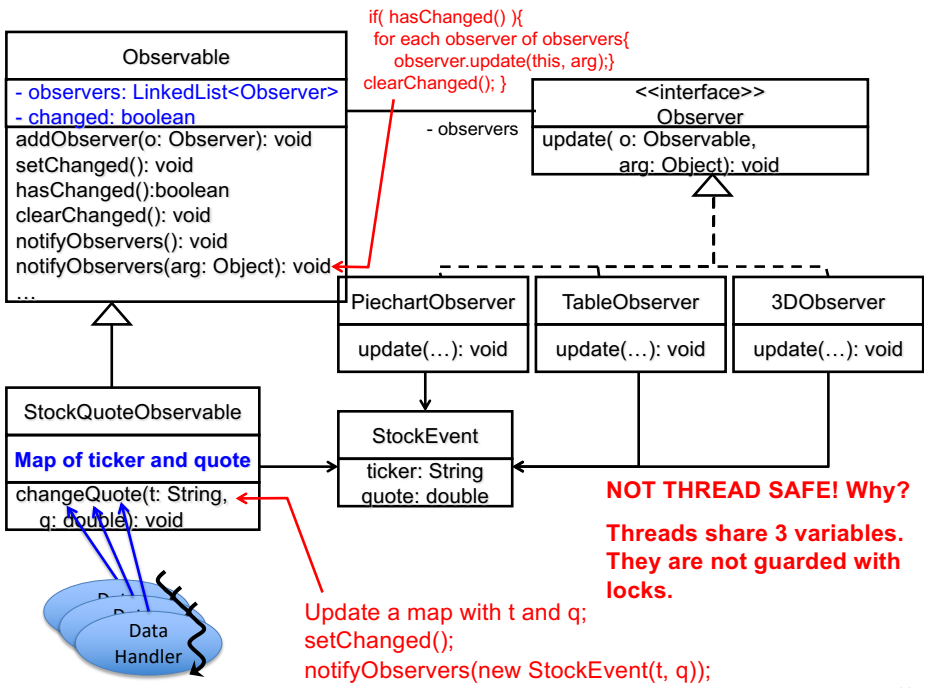




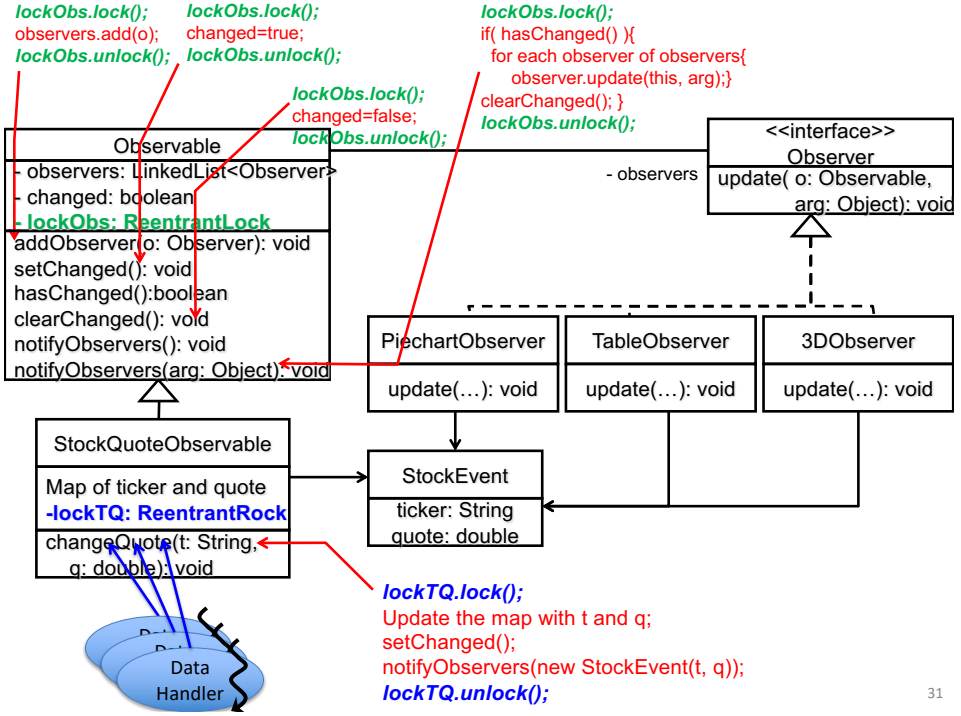


- DataHandler
  - Its instance is responsible for getting financial data from a stock exchange periodically.
  - Defined as a Runnable class.
    - Its run() is executed on a thread.
  - Handles data acquisition concurrently (not sequentially).





30



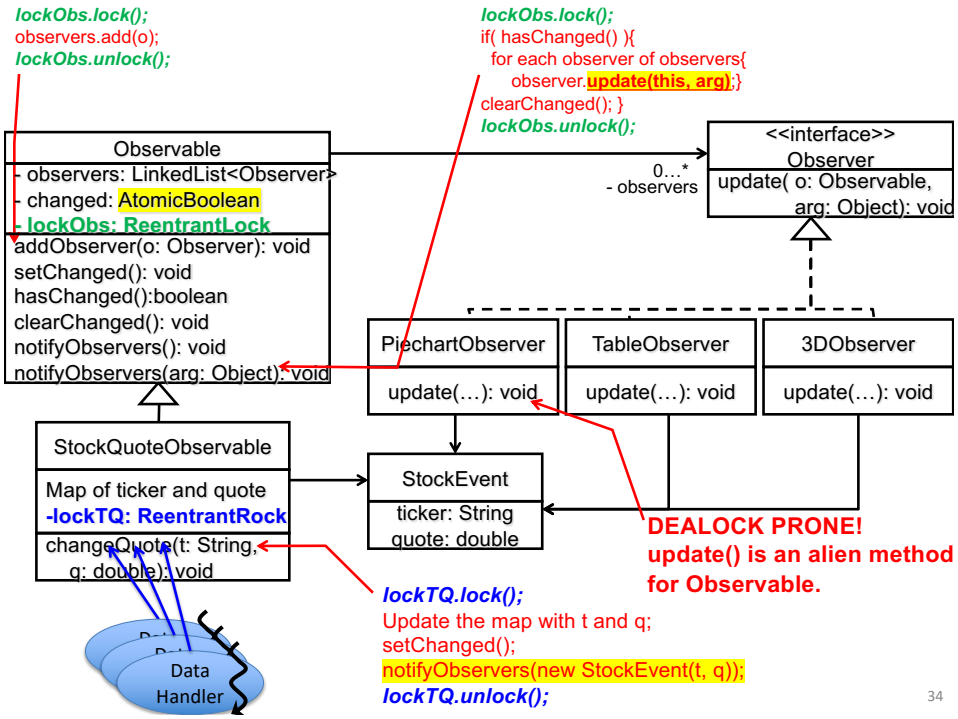
31

Notes

- Each thread acquires two locks in order in `changeQuote()`
  - lockTQ and then lockObs
  - Potential lock-ordering deadlocks?
    - Any threads that try to acquire lockObs and then lockTQ?
    - No, fortunately.
- For simplicity, `Observable` guards two shared variables (`observers` and `changed`) with a single lock.
  - For performance improvement, lock-per-shared-variable policy would be a better choice.
  - Thread synchronization (locking) can be omitted for `changed` by typing it with `AtomicBoolean`.

Observable and Observer

- Thread-safe, but deadlock-prone due to an *alian method call* in `notifyObservers()`.



`Observable.notifyObservers()` in Java API (`java.util`)

- `update()` is invoked as an open call.

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

 public void notifyObservers(Object arg){
   Object[] arrLocal;
   synchronized (this){         // lockObs.lock();
     if (!changed) return;      // balking
     arrLocal = obs.toArray(); // observers copied to arrLocal
     changed = false;
   }                            // lockObs.unlock();

   for (int i = arrLocal.length-1; i < =0; i--)
     ((Observer)arrLocal[i]).update(this, arg); // OPEN CALL
 }
 ……
}
```

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

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

A context switch can occur here, and addObserver() or removeObserver() may be called.

arrLocal may not be in synch with obs. Race conditions can occur.

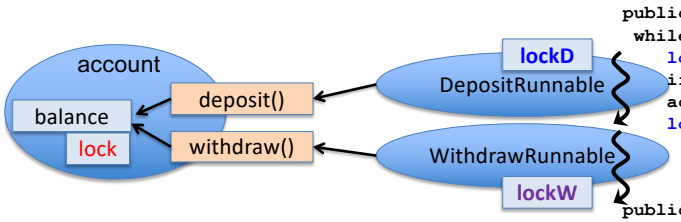
Comments in Java API Doc Say…

- 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).

HW 16

- Use **your own** Observable (class) and Observer (interface).
  - c.f. HW 1
- Your Observable is not thread-safe.
- Revise it to be thread-safe.
  - Use `AtomicBoolean` to define the data field `changed`.
  - Use `ReentrantLock`, not the `synchronized` keyword, to guard a shared data field (i.e. linked list of observers).
  - Do an open call in `notifyObservers()` to avoid potential deadlocks.

Recap: Thread-safe Bank Account



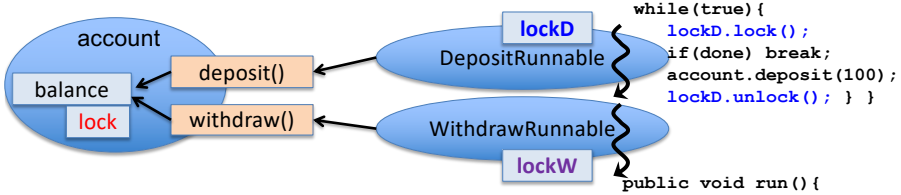

```
public void run(){
 while(true){
    lockD.lock();
 if(done) break;
 account.deposit(100);
 lockD.unlock(); } }
```

```
public void run(){
 while(true){
    lockW.lock();
    if(done) break;
 account.withdraw(100);
 lockW.unlock(); } }
```

```
withdraw(double amount){
     lock.lock();
     while(balance =< 0){
          try{
               // waiting for the balance to exceed 0
               sufficientFundsCondition.await();
          }catch(InterruptedException e){
               break; // Balking
          }
     }
     balance -= amount;
   lock.unlock();  }
```

Each "deposit" thread acquires **lockD** and **lock**.

Each "withdraw" thread acquires **lockW** and **lock**.




```
public void run(){
 while(true){
    lockD.lock();
 if(done) break;
 account.deposit(100);
 lockD.unlock(); } }
```

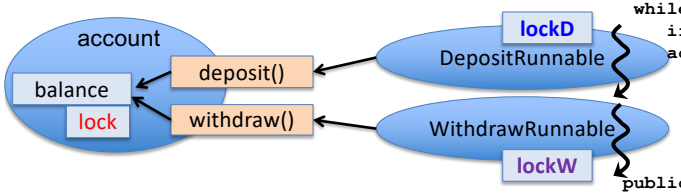
```
public void run(){
 while(true){
    lockW.lock();
    if(done) break;
 account.withdraw(100);
 lockW.unlock(); } }
```

Can a lock-ordering deadlock occur?

No – *as far as the main thread never call* `setDone()` *with BankAccount's* ***lock*** *held.*

In this app, `deposit()` and `withdraw()` must be called with **lockD** and **lockT**, respectively. Open call cannot be a good solution.

`deposit()` and `withdraw()` must be called ONLY WHEN `done!=true`




```
public void run(){
 while(true){
    if(done) break;
 account.deposit(100);} }
```

```
public void run(){
 while(true){
    if(done) break;
 account.withdraw(100);} }
```

Can eliminate a risk of lock-ordering deadlock by

Defining **lockD** and **lockT** with the `volatile` keyword or `AtomicBoolean`