The Problem with Threads

Observer Pattern

- An object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes
- Mainly used to implement distributed event handling systems

Java implementation (1)

```
public class ValueHolder {
    private List listeners = new LinkedList();
    private int value;
    public interface Listener {
        public void valueChanged(int newValue);
    }
    public void addListener(Listener listener) {
        listeners.add(listener);
    }
    public void setValue(int newValue) {
        value = newValue;
        Iterator i = listeners.iterator();
        while(i.hasNext()) {
            ((Listener)i.next()).valueChanged(newValue);
        }
    }
}
```

A list of listeners

State

Register new listener

Value changed

Automatically notify listeners

```
public class ValueHolder {
    private List listeners = new LinkedList();
    private int value;
    public interface Listener {
        public void valueChanged(int newValue);
    }
    public void addListener(Listener listener) {
        listeners.add(listener);
    }
    public void setValue(int newValue) {
        value = newValue;
        Iterator i = listeners.iterator();
        while(i.hasNext()) {
            ((Listener)i.next()).valueChanged(newValue);
        }
    }
}
```

What if multiple threads can call setValue() or addListener() at the same time?

The listeners list gets modified while the iterator is iterating through the list. This will trigger an exception and likely terminate the program

Java implementation (2)

```
public class ValueHolder {
    private List listeners = new LinkedList();
    private int value;
    public interface Listener {
        public void valueChanged(int newValue);
    }
    public synchronized void addListener(Listener listener) {
        listeners.add(listener);
    }
    public synchronized void setValue(int newValue) {
        value = newValue;
        Iterator i = listeners.iterator();
        while(i.hasNext()) {
            ((Listener)i.next()).valueChanged(newValue);
        }
    }
}
```

When a synchronized method is called, the calling thread attempts to acquire **an exclusive lock** on the object.
If any other thread holds that lock, then **the calling thread stalls** until the lock is released.

Possible Deadlock!

Thread 1

```
public synchronized void
addListener(Listener listener) {
        listeners.add(listener);
    }
```

Waiting for Thread 2

```
public synchronized void
setValue(int newValue) {
        value = newValue;
        Iterator i =
listeners.iterator();
        while(i.hasNext()) {

((Listener)i.next()).valueChanged(
newValue);
        }
    }
```

It can do anything in its valueChanged() method, including acquiring a lock

Thread 2

Waiting for another thread, which might be waiting for Thread 1

Final version

```
public class ValueHolder {
    private List listeners = new LinkedList();
    private int value;
    public interface Listener {
        public void valueChanged(int newValue);
    }
    public synchronized void addListener(Listener listener) {
        listeners.add(listener);
    }
    public void setValue(int newValue) {
        List copyOfListeners;
        synchronized(this) {
            value = newValue;
            copyOfListeners = new LinkedList(listeners);
        }
        Iterator i = copyOfListeners.iterator();
        while(i.hasNext()) {
            ((Listener)i.next()).valueChanged(newValue);
        }
    }
}
```

Reference

- Edward A. Lee, The problem with threads, 2006 http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.pdf