

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);
        }
    }
}
```

Annotations:

- A list of listeners (points to `listeners`)
- State (points to `value`)
- Register new listener (points to `addListener`)
- Value changed (points to `setValue`)
- Automatically notify listeners (points to the `while` loop)

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

```
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);
        }
    }
}
```

Annotations:

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

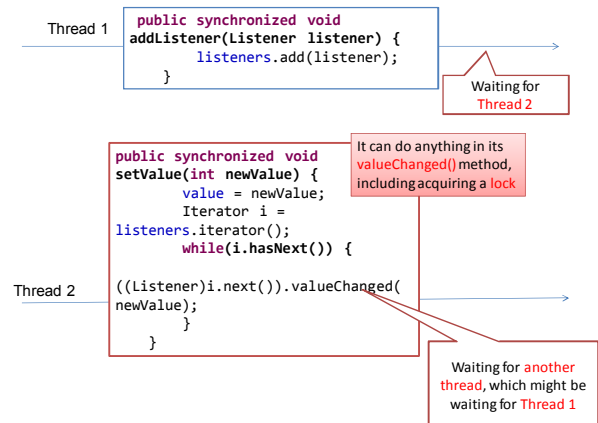
Java implementation (2)

```
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);
        }
    }
}
```

Annotation:

- 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!



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>