Agenda

- Quick review of the Observer pattern
- Worked example
- Exam review (~30 minutes)
- Quiz (on your own time)
Observer Pattern – Problem Statement

- I have some object A whose state (properties) changes from time to time.
- I have some other objects \{B\} which care about the state of A and may wish to take actions when A changes state, e.g.:
  - Update some user interface element
  - Record the change event
  - Notify some other component about the change
  - Etc.
The Observed object (a.k.a. “Subject”) implements some interface (or extends some class) allowing other classes to “register” their desire to receive events from the Observed class.

The Observing objects (Observers) implement a call-back and register themselves with the Observed object.

At runtime, the Observed object will notify the Observers of state changes or other events.
Same pattern, different nouns and verbs

- In Java:
  - Observer/Observable
  - PropertyChangeListener/firePropertyChange
  - Action/ActionListener
  - Other eventing interfaces

- In web applications
  - Subscribe/Notify pattern
  - (related) Publish/Subscribe/Notify

- Most user interface SDKs use a similar model
A callback, in general, is a method in one object that is registered with and invoked by some other object(s)

In the present context, these are generally of the form of update(...), onXXXAction(...), propertyChange(...), etc.

The callback is passed one or more parameters informing it of, e.g.:

- The source of the event
- Metadata about the event, e.g.:
  - The type of event
  - Values related to the event (e.g., old value/new value, which key was pressed, etc.)
Example (from the exercise)

```java
public interface Listenable {
    public void addPropertyChangeListener(PropertyChangeListener listener);
    public void removePropertyChangeListener(PropertyChangeListener listener);
}

public class AnObservableStrokedLine extends AStrokedLine implements Listenable {
    protected PropertyChangeSupport propertyListenerSupport;

    @Override
    public void addPropertyChangeListener(PropertyChangeListener listener) {
        propertyListenerSupport.addPropertyChangeListener(listener);
    }

    @Override
    public void setX(int newX) {
        int old = getX();
        super.setX(newX);
        if (null != propertyListenerSupport) {
            propertyListenerSupport.
                firePropertyChange(PROPERTY_X, old, getX());
        }
    }
}

//Elsewhere...
someLine.addPropertyChangeListener(this);
```
Another Example (from the exercise)

// In some code...

JButton leftButton = new JButton("Left");
leftButton.addActionListener(
    new LineMoverAction(line, MoveDirection.LEFT, MOVE_INCR));

public class LineMoverAction implements ActionListener {

    protected Line line, MoveDirection direction, int increment;

    public LineMoverAction(Line line, MoveDirection direction, int increment) {}  
    @Override
    public void actionPerformed(ActionEvent e) {

        switch (direction) {
            case LEFT:
                line.setX(line.getX()-increment);
                break;
            case RIGHT:
                line.setX(line.getX()+increment);
                break;
            ....
        }
    }
}
Exercise

- Run `Driver.java`; the main application code is in `AppDemo1.java`
- OE shows two lines, but nothing else happens. The buttons don’t do anything either. Hmmph.
- Find POI #1 and #2 around line 49 of `AppDemo1.java` and swap the comments so that we use `AnObservableStrokedLine`.
  - What happens?
- Find POI #8 around line 67 (*idem*) and uncomment the subsequent line.
  - What do happens now?
- Try adding buttons to do other things, e.g.:
  - Move the line up and down
  - Move `line2`
- Check out `LineMoverAction.actionPerformed()` (line 30)
  - Print the output from some of the getters on the `ActionEvent` passed to this callback.
  - Could we figure out whether the event was generated by a mouse click of the button or a keyboard event (space) on the button?