OPI
Lecture 21
Graphical User Interfaces

Kasper Østerbye

Carsten Schuermann

IT University Copenhagen

- Goal:
  - understand the concepts of Graphical User Interfaces
  - understand the fundamentals of event based programming
    - Drawing
    - System events
    - User events
  - understand the fundamentals of Swing
    - (Panels & Layout)
    - Events
    - Delegates
    - Models
  - Enable you to explore Swing or some other GUI library on your own.

Simple Drawing

If you run the program to the right, a small window appears, in which you can draw small dots under the mouse cursor by left clicking the mouse.

Each *component* in a modern windows system has associated a piece of screen on which it can draw. In Swing this bitmap is called a *Graphics*.

A graphics object is the only object through which one can actually print things on the screen.

The full name is java.awt.Graphics

It has methods for drawing

- lines, ovals, and other simple figures
- Texts
- Images

It has a current Font and a current draw color. Both can be changed. Check the API.

```
public class DotFrame extends JFrame {

   public DotFrame(){
      setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      setSize(400, 400);
      setTitle("DotFrame");
      this.addMouseListener(
         new MouseAdapter(){
            public void mouseClicked(MouseEvent e){
               drawDot( e.getX(), e.getY() );
            }
         });
      setVisible(true);
   }

   private void drawDot(int x, int y){
      getGraphics().fillOval(x-5, y-5,10,10);
   }

}
```

Redraw

If the window from before is fully or partially covered, the dots drawn are erased.

A Graphics object has no memory.

It is a direct connection to a part of the screen – but that part might be used by other windows sometimes.

If the graphics object should be able to remember its state, it had to keep a copy of its rendition somewhere else.

It does not do this.

Instead, the Java virtual machine call the *paint(Graphics)* method when our screen is again free.

We will extend to program to have a memory of which dots have been drawn.

There is a class for representing Points, we will use that.

(See DotFrame2.java)

Clipping area

Consider the program to the right.

The only change is that the paint method draws blue dots instead of black ones.

A graphics object has associated with it a "clipping area", which is the area is in need of redraw.

Only draw operations which fall inside this clipping area are really drawn, the others are ignored.

The clipping area is an important mechanism to ensure performance.

```
private void drawDot(int x, int y){
    getGraphics().setColor(Color.BLACK);
    getGraphics().fillOval(x-5, y-5,10,10);
    Point p = new Point(x,y);
    myDots.add( p );
}

public void paint(Graphics g){
    g.setColor(Color.BLUE);
    for(Point p: myDots)
       g.fillOval(p.x-5, p.y-5,10,10);
}
```

(Not) Drawing text

Consider the following drawing task:

- Each time a key on the keyboard is pressed, draw it on the graphics object.

This is done using the code to the right.

However, all letters are drawn on top of each other.

To do this properly, we must be able to:

- Get the size of each letter
- In the font currently used
- And remember all the letters
- And handle delete
- And handle paste

Do not attempt to write on a graphics. One can draw a few Strings – nothing more.

```
// added in the constructor
 this.addKeyListener(
   new KeyAdapter(){
     public void keyTyped(KeyEvent e){
        drawKey( e.getKeyChar() );
     }
  });

 private void drawKey(char c){
    getGraphics().drawString(""+c, 50,50);
};
```

Swing and AWT

GUI building can roughly be divided into two parts

- Drawing on graphics
- Using components

We will now examine to the second part.

A user interface is build from components which can react to user interaction.

Some typical components are:

- Labels
- Text fields
- Buttons
- Radio buttons
- Drop down boxes
- Lists
- Tables

There are three important aspects to consider:

1. Will the user be able to figure out how to work with this interface?
2. How can we build the interface so it looks like we would like it to look?
3. How can we program it to do as we want when the programmer work with the window.

We will ignore 1.

The way in which Swing and AWT handles 2 is embarrassingly complex and unsystematic. There are few general lessons in that.

We will spend the most of the time on 3.

Layout in Swing and AWT

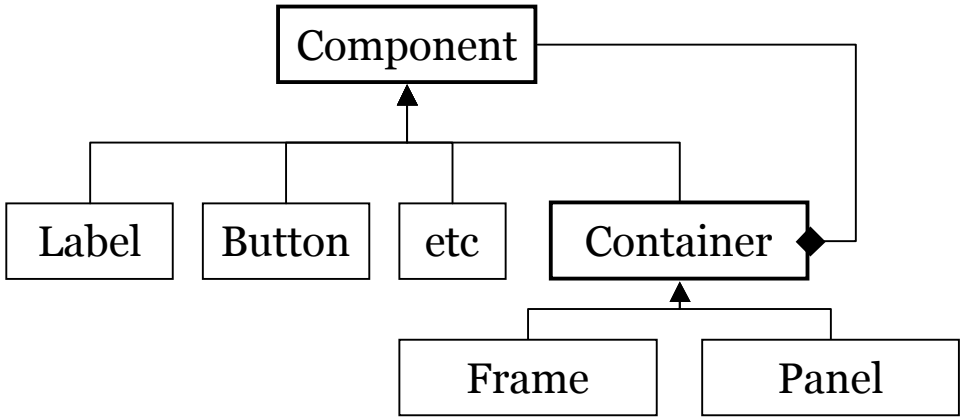
Each component is placed in a container. A container can contain several components.

There exist two kinds of containers:

Frames and Panels

Frames are individual windows on the screen. Panels are areas within the Frame.

A Panel is itself a component.



The physical layout of the components in a container is controlled using a *Layout manager*. There are many different layout managers, all controlled differently.

One positive thing can be said: The design is able to handle resizing of non-trivial user interfaces.

The design above is know as the composite pattern, and was originally invented for this particular purpose.

It is well suited for the arrangement of hierarchical structures.

The clue is that container is itself a component.

The drawing/paint lesson

Each component must be able to paint itself.

It is therefore necessary that it holds all the information necessary to do so.

A component can be seen as consisting of three different parts:

- The model: The data which should be painted.
- The view: The thing you see and the screen.
- The controller: How you interact with the data.

From very early on (1978) these three parts have been split into three different objects.

The pattern is known as Model-View-Controller (MVC).

In Java, the view and control is reunited into one object, called a *Delegate*.

But MVC helps on other issues as well:

Sometimes the same information is shown at the same time at different locations in a user interface.

Sometimes the same information is shown in different ways.

Having a single model keeps the data consistent!

Sometimes the data shown is changed by other means than the GUI.

Changing the model, and updating all views does the trick!

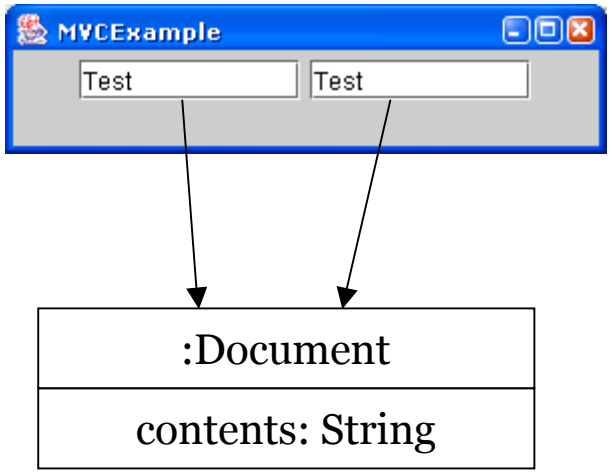
A simple example

The MVCExample to the left has two delegates which uses the same model.

The delegate is the swing JTextField

The model is a Document (from swing as well).

- Any changes done in one delegate, changes the model.
- Any change in the model updates all delegates.



See MVCExample in the lecture code

Observer patten

Sometimes one need to know in object A if there is a change in object B.

This can be done by having B call a method in A.

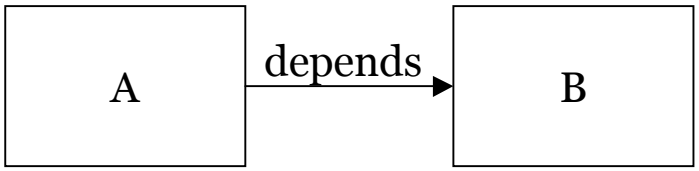
However, this will make B depend on A, which was not the idea.

Solution 1: Make a thread in A, which every 1/10 second checks B to see if there is a change.

Solution 2:

Realise that this situation is so common that special design is necessary.

Solution 2 is the commonly chosen, and is called the *Observer pattern*



Observer and Observable (A and B)

The observable *knows* it is observed, but not by whom.

The observable is designed to be observed. It notifies those observing it about changes.

1. A tells B that it want to observe.
2. B tells all observers that it has changed
3. A gets a message that B has changed
4. A inquires B for its new state

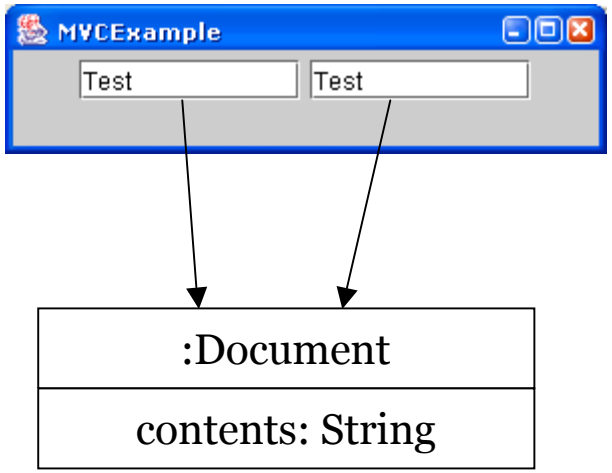
Model is observable

The Document is the Model, a Model is an observable.

The JTextField is an observer.

In Java GUI framework, *observers* are sometimes called **listeners**.

Here, the synchronization happens simply because it is the same model.




```
JTextField tf1 = new JTextField(10);
Document document = tf1.getDocument();
JTextField tf2 = new JTextField(document,null,10);
```

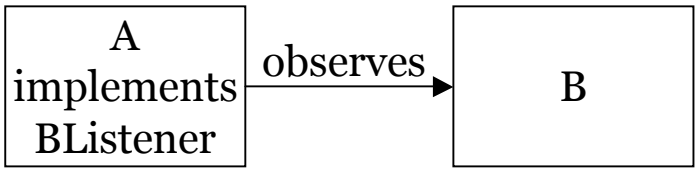
Observables and Listeners

When an Observable notifies its observers, it will assume that the observers/listeners implement a specific ListenInterface.

The listen interface is different for different observables.

There is a division of labor between the programmer of A and B.

1) B knows *when* something interesting is happening, but *not what* to do.
2) A knows *what* to do, but *not when*.



1. A tells B it is interested in what B is doing, typically as:
   myB.addBListener(myA);
2. The B object does something which changes it state. It then calls a method M from the BListener interface on all objects which has been added as listeners.
3. The M method on myA is called. What happens here is up the the A programmer.

A document listener

The code to the right is a label, which show how long a document is.

Question: How do one write such a thing?

Problem: How did I find out I needed a Document listener, and what must I implement

1) JTextField's superclass has massive documentation in API – which mention 'Document'
2) getDocument brings me to Document.
3) Document has addDocumentListener
4) DocumentListener has three methods which must be implemented.
5) I find out that I do not need to know about DocumentEvent

Try to follow the above steps yourself.

```
class DocumentSizeView
    extends JLabel
    implements DocumentListener {

    Document d;
    DocumentSizeView(Document d){
        super("");
        this.d = d;
        setText("Size: " + d.getLength() );
        d.addDocumentListener(this);
    }

    public void changedUpdate(DocumentEvent e) {}
    public void insertUpdate(DocumentEvent e) {
        setText("Size: " + d.getLength());
  }
    public void removeUpdate(DocumentEvent e){
        setText("Size: " + d.getLength());
    }
}
```

Changing the document

The class to the right inserts 'HaHa' into a document every 3 second.

Notice, that when the program is run, the listeners are updated to reflect the changes to the model.

```
class InsertHaHa implements Runnable {
    Document d;
    InsertHaHa(Document d){
        this.d = d;
    }
    public void run(){
        int start = d.getStartPosition().getOffset();
        while(true){
            try{
                Thread.sleep(3000);
                d.insertString(start, "HaHa", null);
            }catch(Exception ignore){};
        }
    }
}
```

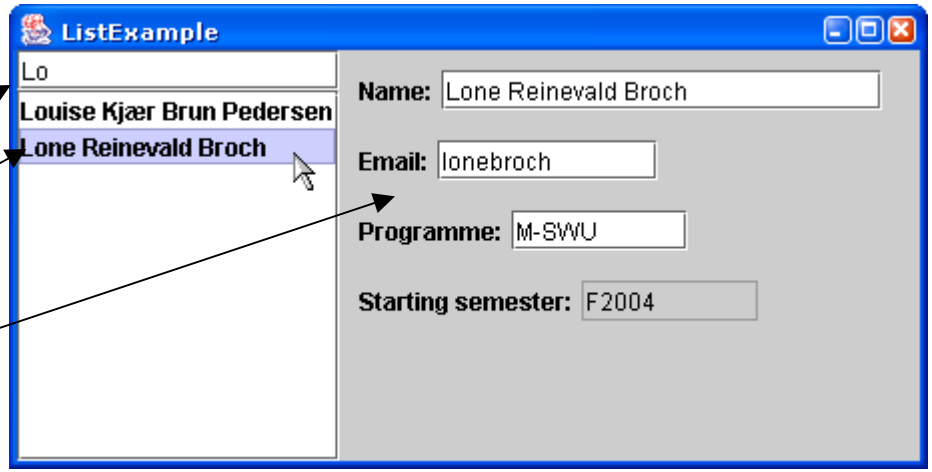
A much larger example

(for reading and study –
I do not expect you to write such a thing)

Selecting and editing a studen list

The GUI to the right has three main components:

- A text field which serves as a filter
- A list from which one can select a student
- An area in which the data of the selected student is shown.



When the filter field is changed, the list shrinks or expands (shrinks as characters are written to the field, expands as they are deleted)

Selecting a student brings up the data of that student. If the filter reduces the list to exactly one element, that element is automatically selected.

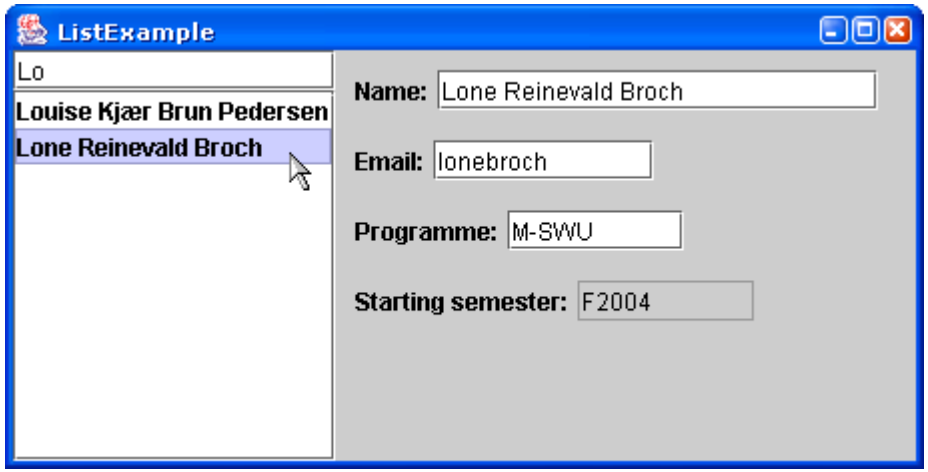
Editing the data of the student name is immediately reflected in the list.

We shall look at the list and its model, and we will examine how changing a name can update the list.

And we shall try to do this in a modular way.

The top level window

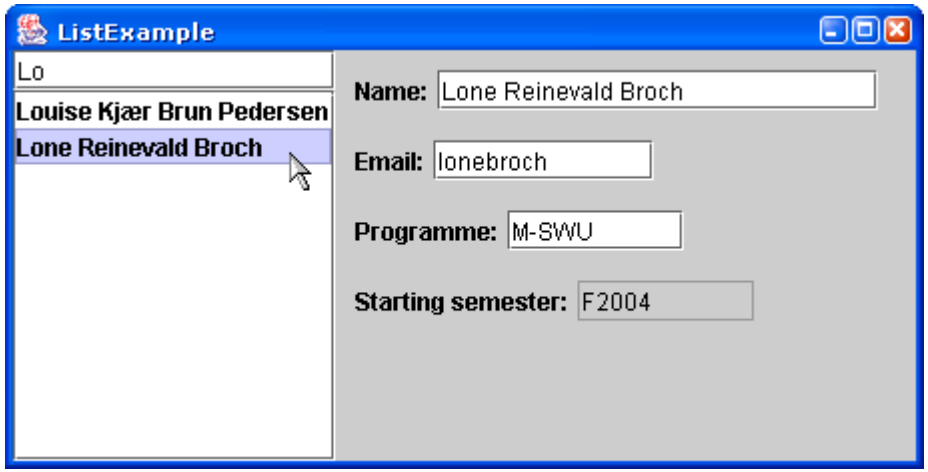
```
class MyWindow extends JFrame {

    MyWindow(){

setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(700, 400);
        setTitle("ListExample");

        final SelectingList students =
                new SelectingList(OOPStudent.students);
        this.getContentPane().add( students, "West");

        final OOPStudentDelegate sd =
                        new OOPStudentDelegate();
        this.getContentPane().add( sd, "Center");

        setVisible(true);
    }
}
```




```
students.addListener(
    new SelectingList.Listener(){
        public void selectionChanged
            (SelectingList.Listener.Event ev){
                if (ev.getSelection() == null)
                    sd.setModel(null);
                else
                    sd.setModel (
                    (OOPStudent)ev.getSelection() );
        }});
```

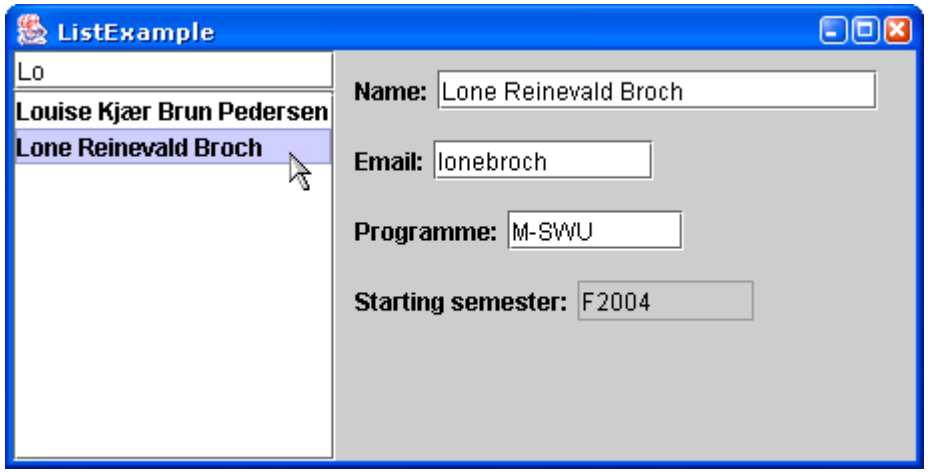
The seleting list

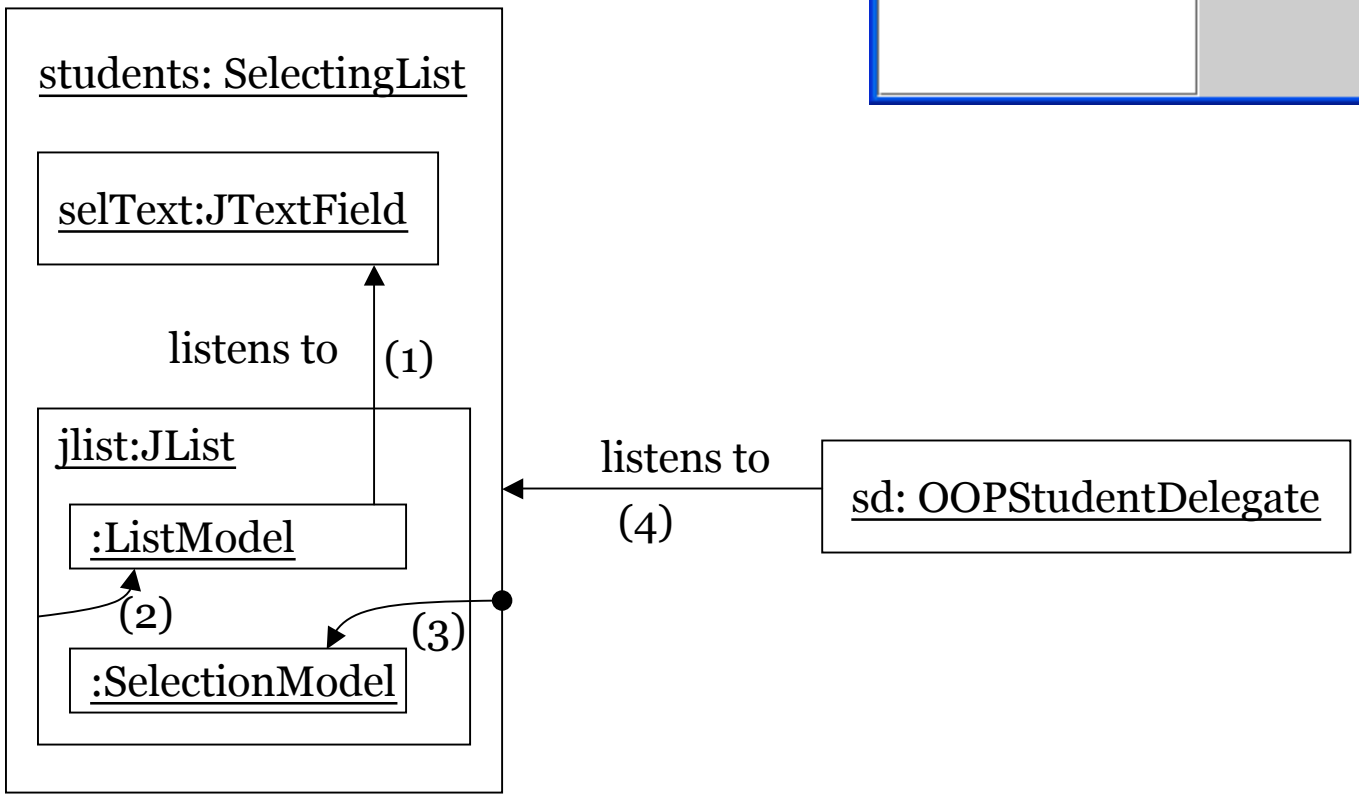
```
public class SelectingList extends JPanel{
    private final JList jlist;
    private final SelectingListModel listModel;
    JTextField selText;

    public SelectingList(Object[] elements){
        listModel =
            new SelectingListModel( elements );
        jlist = new JList( listModel );
        jlist.setSelectionMode(
            ListSelectionModel.SINGLE_SELECTION );
        …
        selText.addCaretListener( new CaretListener(){
            public void caretUpdate(CaretEvent e){
                listModel.updateFilter( selText.getText() );
                if ( listModel.getSize() == 1 )
                    jlist.setSelectedIndex(0);
                else
                    jlist.clearSelection();
            }
        });
```




```
        jlist.addListSelectionListener(
            new ListSelectionListener(){
                public void valueChanged(ListSelectionEvent e){
                    if (! e.getValueIsAdjusting() )
                        if ( jlist.isSelectionEmpty() )
                            notifyListeners( null );
                        else
                            notifyListeners(
                                listModel.getElementAt(
                                    list.getMinSelectionIndex() ) );
                }
        });
    }
```

Listen structure





SelectingListModel

The SelectingListModel is the model we make to be used by JList. It must satisfy the ListModel interface:

```
Object getElementAt(int index)
int getSize()
void addListDataListener(ListDataListener l)
removeListDataListener(ListDataListener l)
```

The abstract class AbstractListModel implements the listener aspects, and gives us the method:

```
void fireContentsChanged
            (Object source, int index0, int index1)
```

which we can use to tell our dependents that we have changed.

```
private static class SelectingListModel extends
AbstractListModel {
    private final Object[] rawData;
    private java.util.List filteredData = new ArrayList();
    private String filter;
    SelectingListModel(Object[] raw){
        rawData = raw;
        updateFilter("");
    }
    private boolean checkAgainstFilter(Object o){
        …
    }
    void updateFilter(String newFilter){
        …
        this.fireContentsChanged
                ( this, 0, filteredData.size() );
    }
    public int getSize(){ …}
    public Object getElementAt(int index){…}
}
```

update the filter

The list filteredData is cleared, and we check all elements in rawData to find those which match the filter.

```
void updateFilter(String newFilter){
        filter = newFilter;
        filteredData.clear();
        for (int i = 0; i<rawData.length; i++)
            if ( checkAgainstFilter( rawData[i] ) )
                filteredData.add(rawData[i]);
        this.fireContentsChanged
            ( this, 0, filteredData.size() );
}
```

The OOPStuden class

This class is build as a model. It is a subclass of Observable:

void addObserver(Observer o)

protected void clearChanged()

int countObservers()

void deleteObserver(Observer o)

void deleteObservers()

boolean hasChanged()

void notifyObservers()

void notifyObservers(Object arg)

If this object has changed, as indicated by the hasChanged method, then notify all of its observers and then call the clearChanged method to indicate that this object has no longer changed.

protected void setChanged()

Marks this Observable object as having been changed; the hasChanged method will now return true.

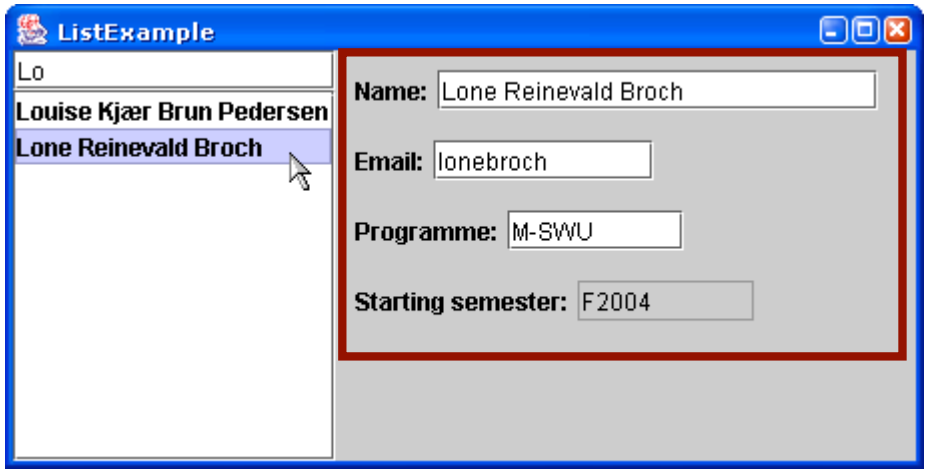
```
public class OOPStudent extends Observable{
    private String name, email,programme,start;
    OOPStudent(String name, String email,
                        String programme, String start){
        this.name = name;   …
    }
    private void notify(String subject){
        setChanged();
        notifyObservers(subject);
    }
    public String getName(){ return name; }
    …
    public void setName(String s){
        name = s; notify("name");}
    …
    public String toString(){return name;}
```

The OOPStudentDelegate

The Delegate is made up of four labels and four text fields.

The starting semester is read only.

Internally, there is a caretlistener on each text field which updates the model as we are typing.



Remember, the Delegate depends on the selection in the list.

```
students.addListener( new SelectingList.Listener(){
    public void selectionChanged
        (SelectingList.Listener.Event ev){
            if (ev.getSelection() == null)
                sd.setModel(null);
            else
                sd.setModel
                    ( (OOPStudent)ev.getSelection() );
    }});
```

```
public void setModel(OOPStudent s){
    model = s;
    if (s == null){
        nameField.setText("");
        nameField.setEditable(false);
        …
    }else{
        nameField.setText( model.getName() );
        nameField.setEditable(true);
        …
    }
}
```

SelectingList Listener

The interface is an *local interface* of class SelectingList. It has one method selectionChanged.

Often the listener methods has an argument which inform about the event that took place.

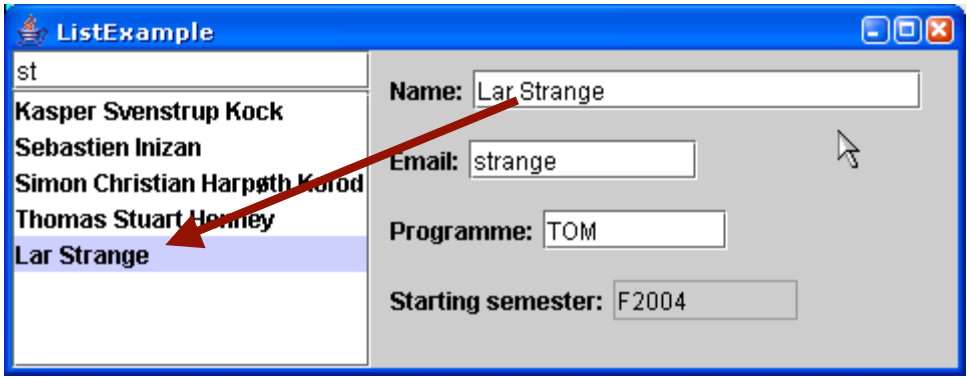
Here I specify the event class as an *local class* in the interface.

By convention, event classes must be subclasses of EventObject.

```
interface Listener {
    class Event extends EventObject{
        private Object selection;
        private Event(SelectingList sl, Object sel){
            super(sl);
            selection=sel;
        }
        public Object getSelection(){
            return selection;
        }
    }

    void selectionChanged(Event e);
}
```

Making the list listen to the students

```
void updateFilter(String newFilter){
    filter = newFilter;
    Iterator itr = filteredData.iterator();
    while (itr.hasNext()){
        Object o = itr.next();
        if (o instanceof Observable)
            ( (Observable) o).deleteObserver(this);
    }
    filteredData.clear();
    for (int i = 0; i<rawData.length; i++)
        if ( checkAgainstFilter( rawData[i] ) ){
            filteredData.add(rawData[i]);
            if (rawData[i] instanceof Observable)
                ((Observable)rawData[i])
                                .addObserver(this);
        }
    this.fireContentsChanged( this, 0, filteredData.size() );
}
```

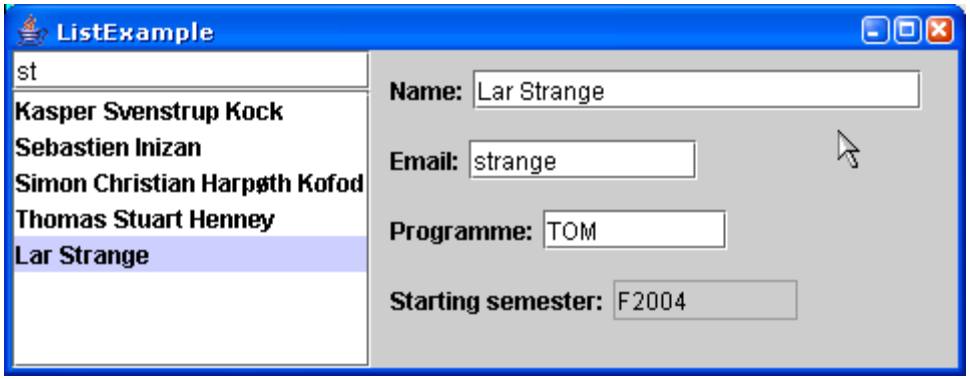


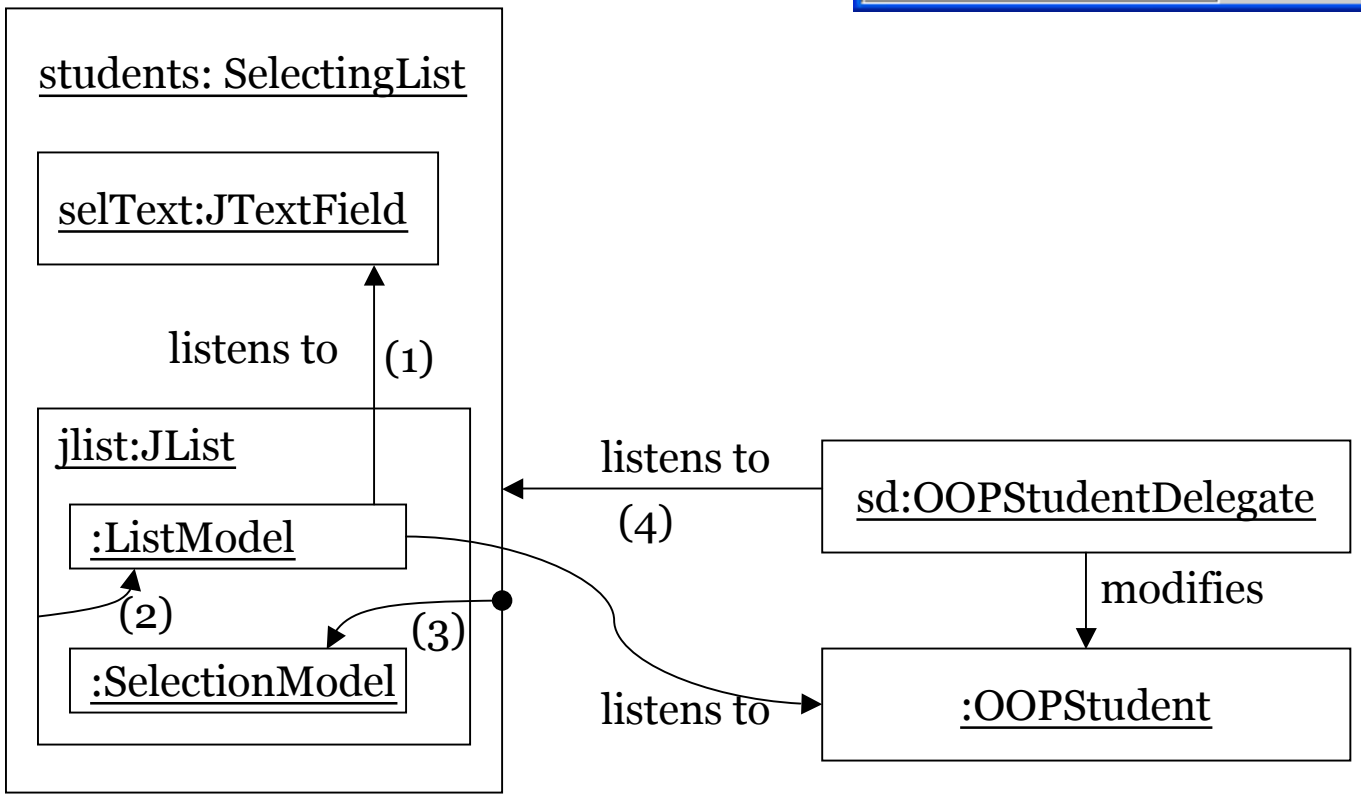
Changing the name of the selected student is immediately reflected in the list.

This is done by making the list listen to all its elements.

When the filter changes, we unsubscribe to all the previous elements, clear the list, and subscribe to all the new elements.

In summary


ListExample
st
Kasper Svenstrup Kock
Sebastien Inizan
Simon Christian Harpøth Kofod
Thomas Stuart Henney
Lar Strange
Name: Lar Strange
Email: strange
Programme: TOM
Starting semester: F2004


students: SelectingList
selText:JTextField
listens to
(1)
jlist:JList
:ListModel
(2)
(3)
:SelectionModel
listens to
(4)
sd:OOPStudentDelegate
modifies
listens to
:OOPStudent

What to make of all this

- Observable/Observer design pattern
- Listeners is one way of doing this.
- Model/View/Controller is in Java Model/Delegate
- Often one will write the model class to reuse the Delegate.

- GUI frameworks has matured over 25 year.
- Swing is designed to be very flexible, at the cost of being highly complex.
- The essence is to figure out
  - What is the model(s)
  - What is the delegate(s)
  - What listeners are available
    - In the delegate
    - In the model

Some alternatives

The listener idea is nearly universal in GUI frameworks today.

However, the concrete design varies from one programming language to the next:

In Java swing :

```
selText.addCaretListener( new CaretListener(){
    public void caretUpdate(CaretEvent e){
        … do stuff…
    }
});
```

In non_swing Java (using reflection)

```
selText.addCaretListener(this, ”doStuff”);
…
public void doStuff(CaretEvent e){
    …
}
```

In C# (not real syntax, but in principle)

```
selText.addCaretListener(
    new Method(CaretEvent e){
        do stuff
    });
```

In some research languages, one do not use the listener idea, but overides a method in the delegate directly.

```
selText = new JTextField(…){
    public void carretUpdate (CaretEvent e){
        … do stuff…
    }
};
```

Alternative layout

```
Frame pm = new Frame("Person Manager") {
    @Vertical
    Panel listpanel = new Panel() {
        @Width(150)
        TextField searchtextfield = new TextField();
        @Width(150) @Height(300)
        List list = new List();
    };
    @Vertical    @Padding(0)
    Panel infopanel = new Panel() {
        @Horizontal
        Panel namepanel = new Panel() {
            @Width(100)
            Label namelabel = new Label("Name:");
            @Width(200)
            TextField nametextfield = new TextField();
        };
    @ Horizontal Panel addresspanel = new Panel() {
        @Width(100)
        Label addresslabel = new Label("Address:");
        @Width(200)
        TextField addresstextfield = new TextField();
    };
    @ Horizontal @Hlock(false)
    Panel phonepanel = new Panel() {
        @Width(100)
        Label phonelabel = new Label("Phone:");
        @Width(200)
        TextField phonetextfield = new TextField();
    };
    @ Horizontal @Hlock(false)
    Panel addpanel = new Panel() {
        IButton removebutton = new Button("Remove");
        IButton addbutton = new Button("Add");
    };
  };
};
```