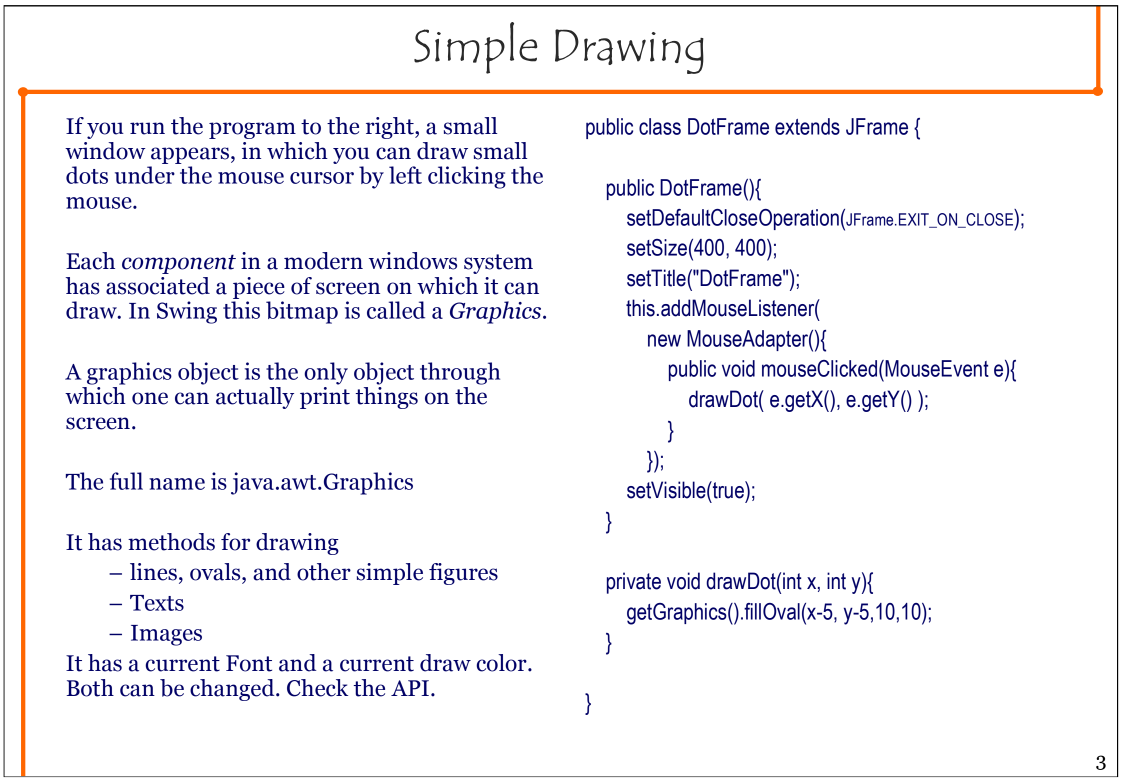
OOP Spring 2005
Lecture 9
Graphical User Interfaces

Kasper Østerbye
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);
  }

}
```

3

A component in a graphical user interface is a very general concept. In Java, it is defined as:

A *component* is an object having a graphical representation that can be displayed on the screen and that can interact with the user. Examples of components are the buttons, checkboxes, and scrollbars of a typical graphical user interface.

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.

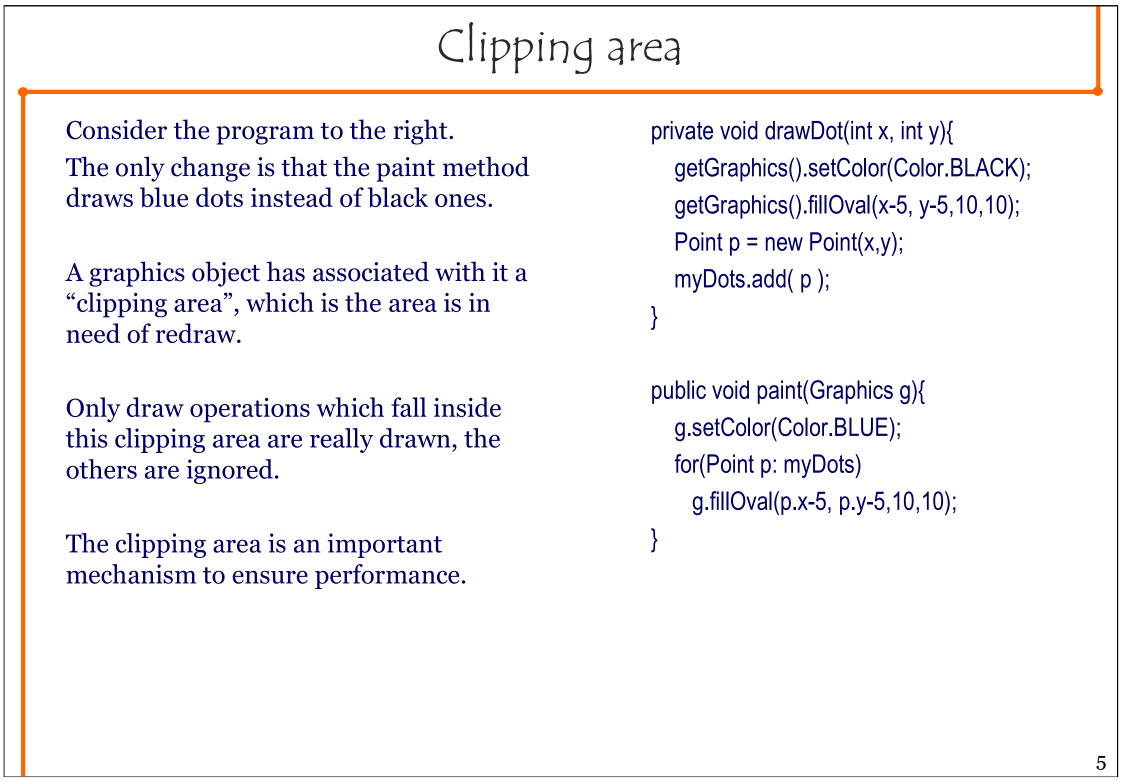
Instead, the Java virtual machine call the *paint(Graphics)* method when our screen is again free.

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)

4



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);
}
```

5

You can notice that the for loop in the paint method attempt to paint all dots, but only succeeds with those within the clipping area.

The graphics object has a method boolean hitClip(x,y, width, height), which returns true if some of the rectangle specified by the parameters are within the clipping area.

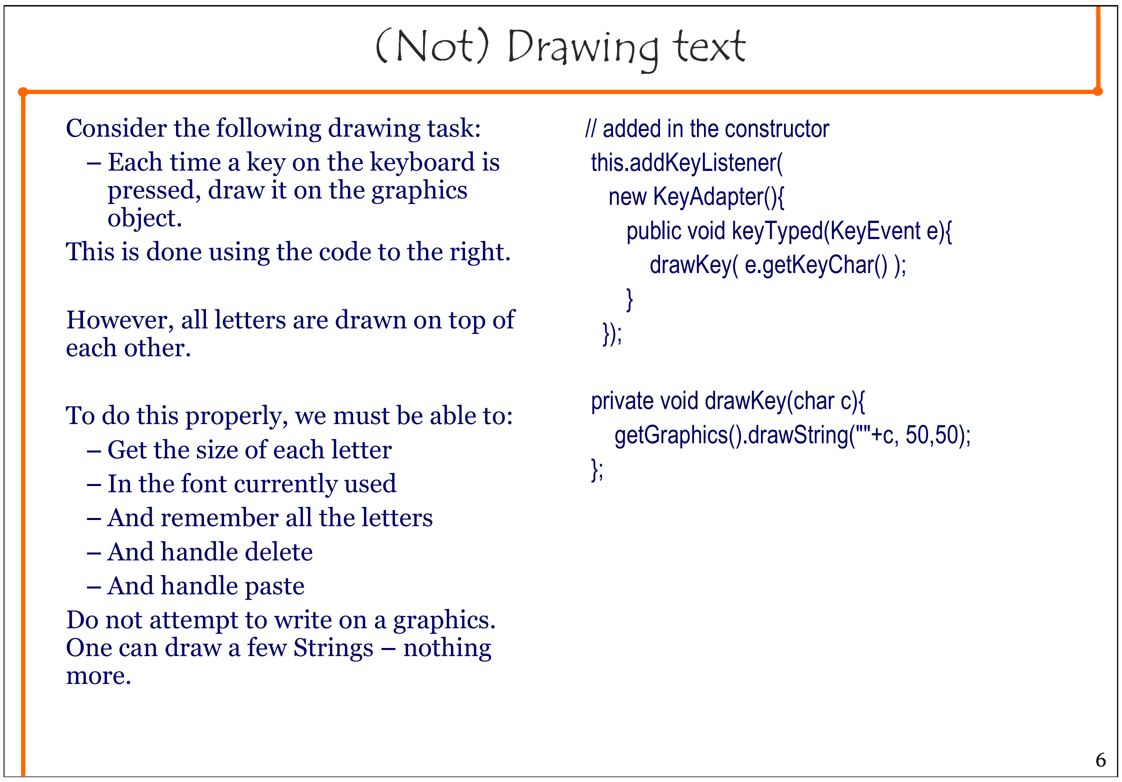
This can be used to rewrite the for loop to the following:

```
 for(Point p: myDots)
    if (g.hitClip(p.x-5, p.y-5,10,10) )
        g.fillOval(p.x-5, p.y-5,10,10);
```

So we only draw if there is a need.

When building large and complicated drawings – for instance large diagrams – it is important to consider the clipping area to obtain reasonable performance when working with windows.

The page http://java.sun.com/products/jfc/tsc/articles/painting/index.html explains about painting in lots of details.



(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);
 };
```

6

Fonts and fonts rendering, and dealing with text which have different font, for instance headings and italics, is a major task. There are lots of classes for doing this in Swing, but it is rather complex. The complexity stems from the fact that we as humans have worked with text for many centuries, and the conceptual models behind typography has now been extended to also include interaction with text.

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

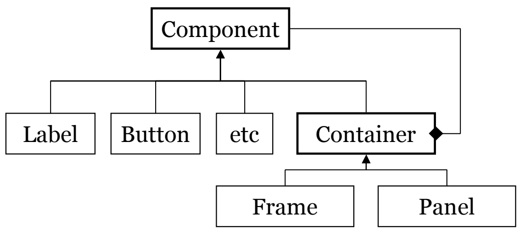
Each component is placed in a container. A container can contain several components.

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.

8

The composite design pattern is widely used. The composite part is in the Swing case the container. The regular components like Label and Button are called the Leafs or Atoms because there are nothing inside them.

There is a small irregularity (which is common in the use of the composite pattern), which is that the Frame is really not a component, a component is something which can be put inside a Container, one can not have Frames within Frames in Swing.

Thomas Quistgaard wrote a masters thesis spring 2005 in which he solved the layout issue in a much nicer way than done in regular Swing. Read it and be enlightened.

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!

9

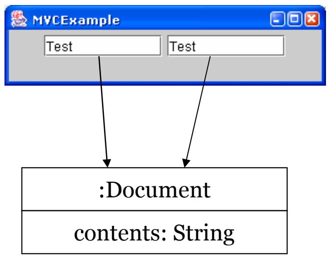
A simple example

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

The delegate is the swing JTextField

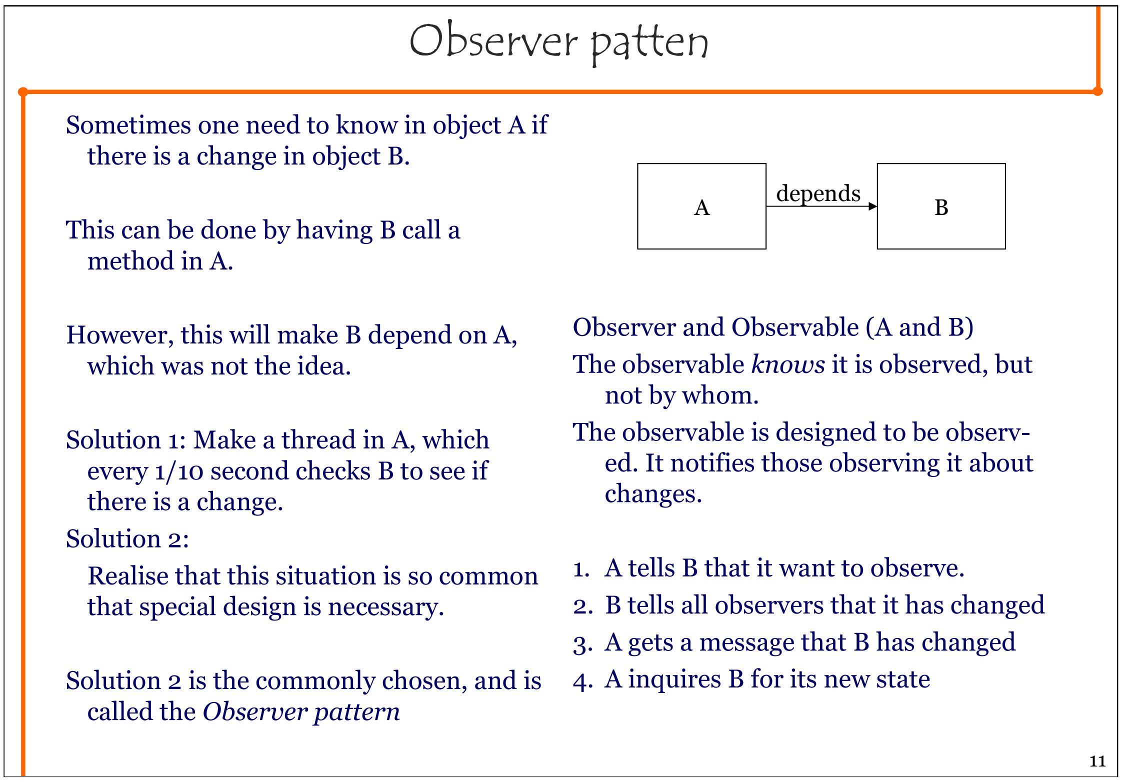
The model is a Document (from swing as well).

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



See MVCExample in the lecture code

The contents of a Document object is actually not a String, as a document also contains information about fonts and other information. However, that will complicate matters unnecessarily.



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.

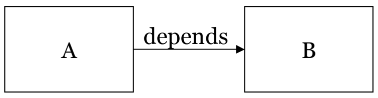
However, this will make B depend on A, which was not the idea.

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 observ-ed. 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

11

Problem with solution 1: What if changes occur every 1/1000 of a second, or once every 10 second.

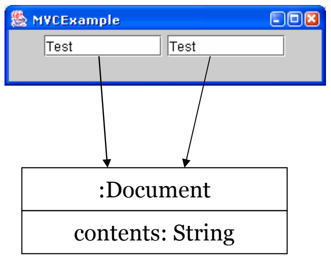
Model is observable

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.



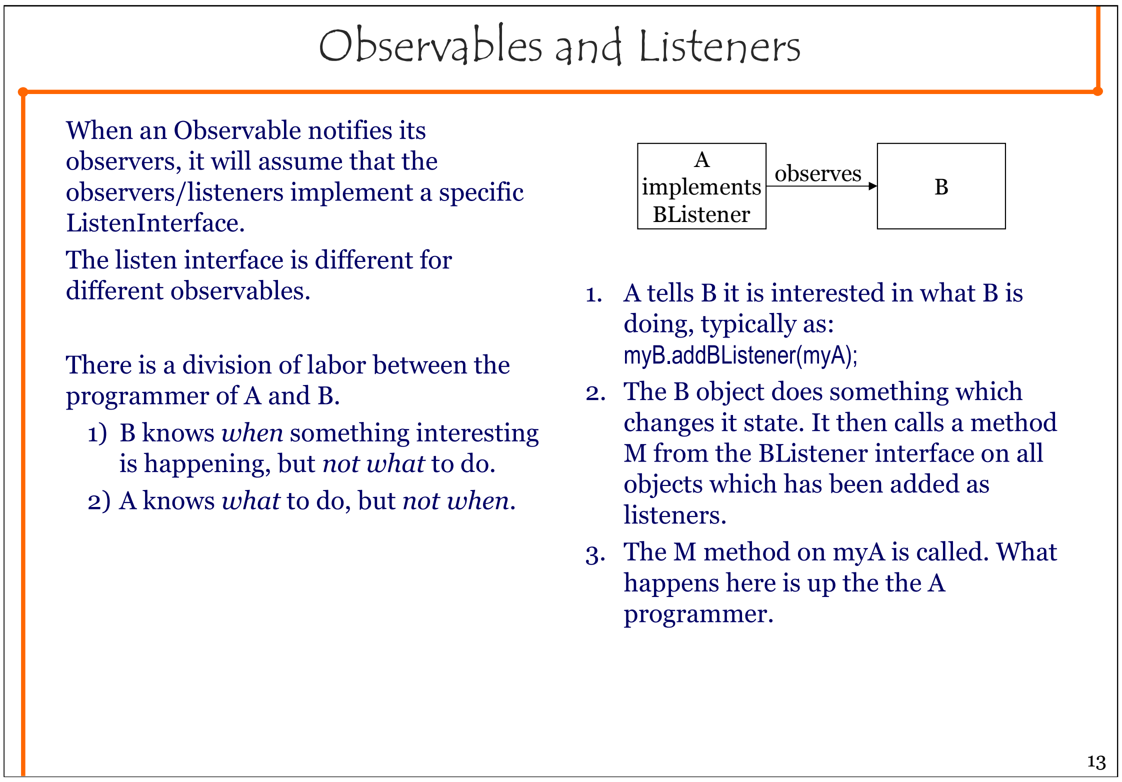
```
JTextField tf1 = new JTextField(10);
Document document = tf1.getDocument();
JTextField tf2 = new JTextField(document,null,10);
```

12

When a JTextField is created it will create a new Document object to keep the text in the field.

Next, we get a reference to the document inside tf1.

Finally, we make the next JTextField use the same document. The null parameter could have been used to specify typeface information.



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*.

$$\begin{tikzcd} \boxed{\text{A implements BListener}} \arrow[r, "\text{observes}"] & \boxed{\text{B}} \end{tikzcd}$$

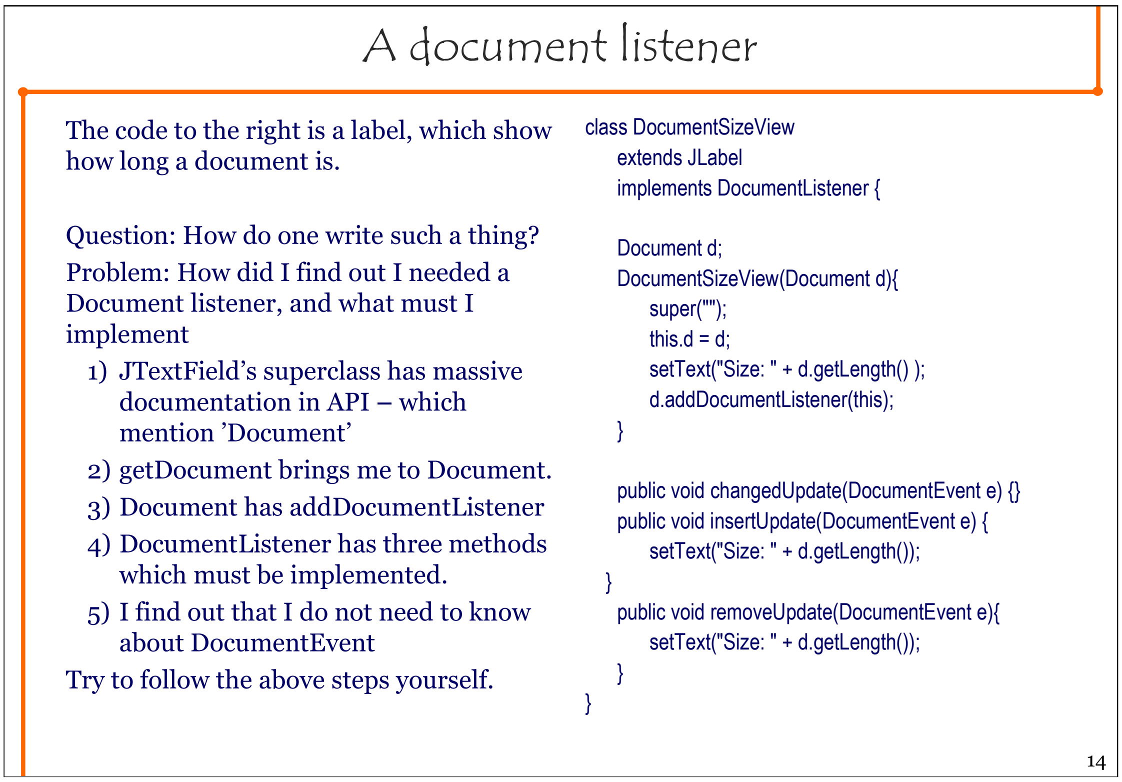
1. A tells B it is interested in what B is doing, typically as: `myB.addBListener(myA);`
2. The B object does something which changes it state. It then calls a method M from the BListener interface on all objects which has been added as listeners.
3. The M method on myA is called. What happens here is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

13

Notice that this division of labour is somewhat similar to what was said of exceptions. The reason is somewhat the same, namely that the programmer of some component (e.g. a Button or TextField) know how such a component should be drawn on the screen, and when the button is pressed or the text is edited.

But the programs to use the button and textfields have not yet been written. Some mechanism must therefore be established which allow the general component (the button) to call functionality in the application (your program).

The designer of the button (or any other GUI component) tries to identify which events can be of interest to the application programmer. For the Button, this is clicking it, but you can also be notified when someone moves the cursor onto the button, or when it leaves it. One can also be notified about keyboard events on the button.



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.

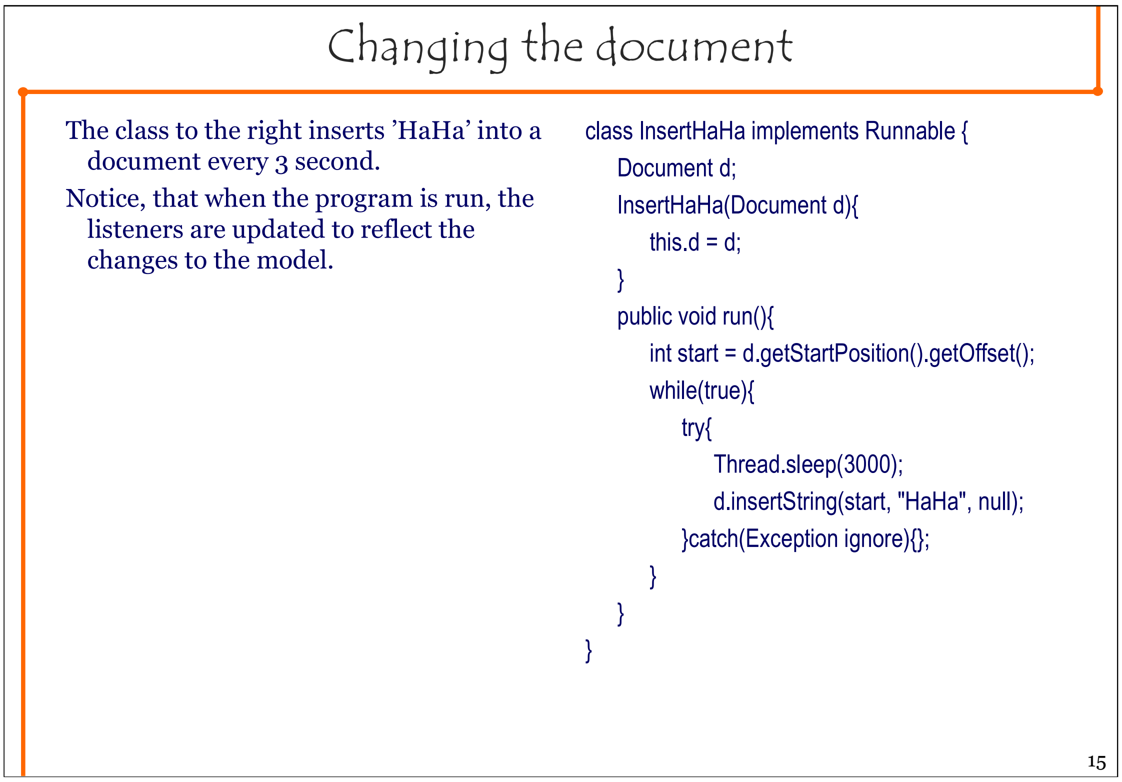
```
class DocumentSizeView
    extends JLabel
    implements DocumentListener {

    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());
    }
}
```

14

It is necessary to ignore information one do not need. The documentation talks about undo handling, documents structured into sections, and properties. However, I am only interested in knowing when the text changes.

The DocumentListener interface specify three methods, changedUpdate, insertUpdate, and removeUpdate. To figure out that I do not need to do anything in the changedUpdate method I read the documentation, which talk about attributes, which I do not know what is. I added a system.out.println to the method to see if I could figure out when it was called, but it never was, so I figure I do not need it.

You will be in the same situation often: Is this method one I should worry about or not. Try to understand the documentation in the API, and make an example program to figure out what happens.



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){};
        }
    }
}
```

15

InsertHaHa Get a Document as parameter to its constructor. Remember that a Document is the model for a JTextField; changes to the model is therefore reflected in the text field.

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

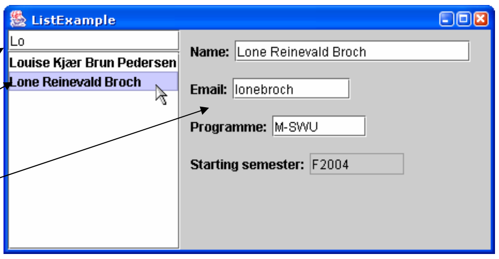
– A list from which one can select a student

– 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.



Lo

Louise Kjær Brun Pedersen

Lone Reinevald Broch

Name: Lone Reinevald Broch

Email: lonebroch

Programme: M-SWU

Starting semester: F2004

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 source code for this example consists of the files:

ListExample.java – the main method of the program

SelectingList.java – the list and filter

OOPStudent.java – representation/model of a student

OOPStudentDelegate.java – presentation of a single student

VerticalFlowLayout.java – a layout manager which places its components vertically

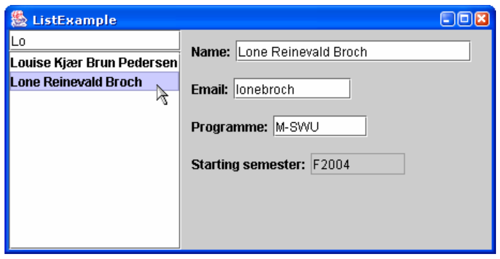
Some of these files contain several classes.

These classes will also be the topic of the exercises.

The goal of examining this example in detail is to show the inner workings of a non-trivial example. There are many interdependent components.

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);
  }
}
```



Lo
Louise Kjær Brun Pedersen
Lone Reinevald Broch

Name: Lone Reinevald Broch
Email: lonebroch
Programme: M-SWU
Starting semester: F2004

```
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() );
      }});
```

18

There are really just two components here, a SelectingList and a OOPStudentDelegate.

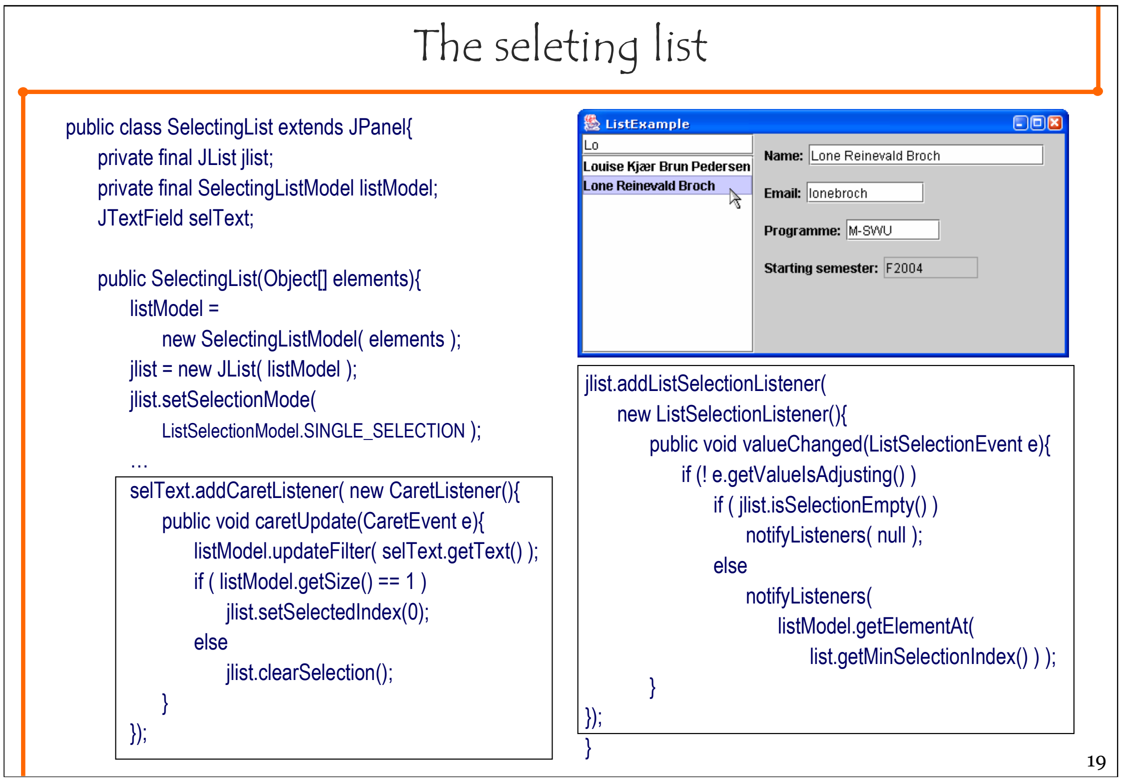
The SelectingList have been designed to be usable for a wide range of lists, whereas the OOPStudentDelegate is a special purpose window which presents information for a specific class (OOPStudent).

The idea of SelectingList is to provide an example of a reusable component which we write ourselves.

The OOPStudentDelegate is not likely to be reusable in other applications, but by making it a component, it is easier to maintain the program. If one need later to edit students, we have a component which can be used for this, and if we need to do a different layout, the OOPStudentDelegate can be inserted a different place.

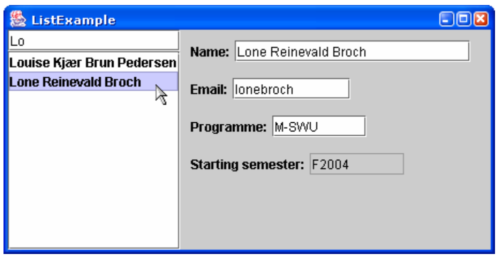
The SelectingList declares a listener so we can be notified when the selection changes (either a new student is selected, or the selected student is deselected).

The listener checks to see if the selection is null (indicating de-selection), or not (indicating selection). Both cases are used to set the student delegate.



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();
            }
        });
```



Lo

Louise Kjær Brun Pedersen

Lone Reinevald Broch

Name: Lone Reinevald Broch

Email: lonebroch

Programme: M-SWU

Starting semester: F2004

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

19

The SelectingList is a class that represents the list **and** the filter. It is made a sublcass of JPanel, which allow us to have the filter field and list inside.

The SelectingList has two GUI parts, the textfield selText, which keeps the selectingText or filter, and the JList, which is the swing component which actually contains the list.

When the selText field changes, the list to be shown changes. We have two ways to do this, either we change the list data based on the filter, or we make our own list model. I have chosen the last solution to illustrate how that is done.

The class SelectingListModel is described on one of the next slides.

There are two Listeners in the above code.

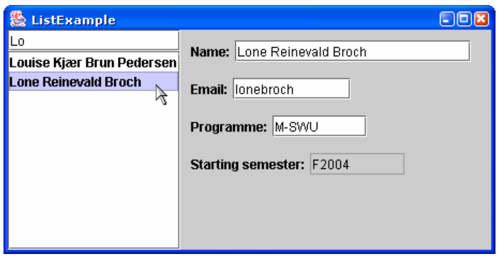
The first listener is a caret listener (caret is one name for the blinking cursor or vertical bar which is indicating where text will be inserted in a text field). The listener will be notified each time a change occurs in the selText field.

The logic is that the selecting list model is told to readjust to the new filter. Then we check to see if the model has exactly one element, and if so, that one element is selected, else no element should be selected.

The second listener is somewhat more complicated. We listen to changes to the selection in the jlist. The first thing to handle is that the jlist will send notifications if one moves the mouse over the list with the left mousebutton down. This is something which is used when the list is set up to allow multiple elements to be selected (which we do not – see setSelectionMode). We can ask the event parameter if the value (selection) is stable or not. We are only interested if it is stable.

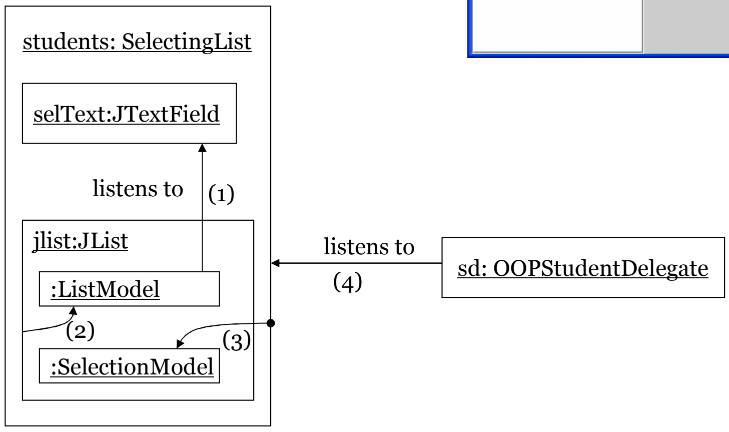
If the jlist does not have any selection, we notify the listeners on SelectionList that there is no selection, otherwise we notify that there is a selection, and what element was selected. The details of this will be on one of the next slides.

Listen structure



Lo
Louise Kjær Brun Pedersen
Lone Reinevald Broch

Name: Lone Reinevald Broch
Email: lonebroch
Programme: M-SWU
Starting semester: F2004



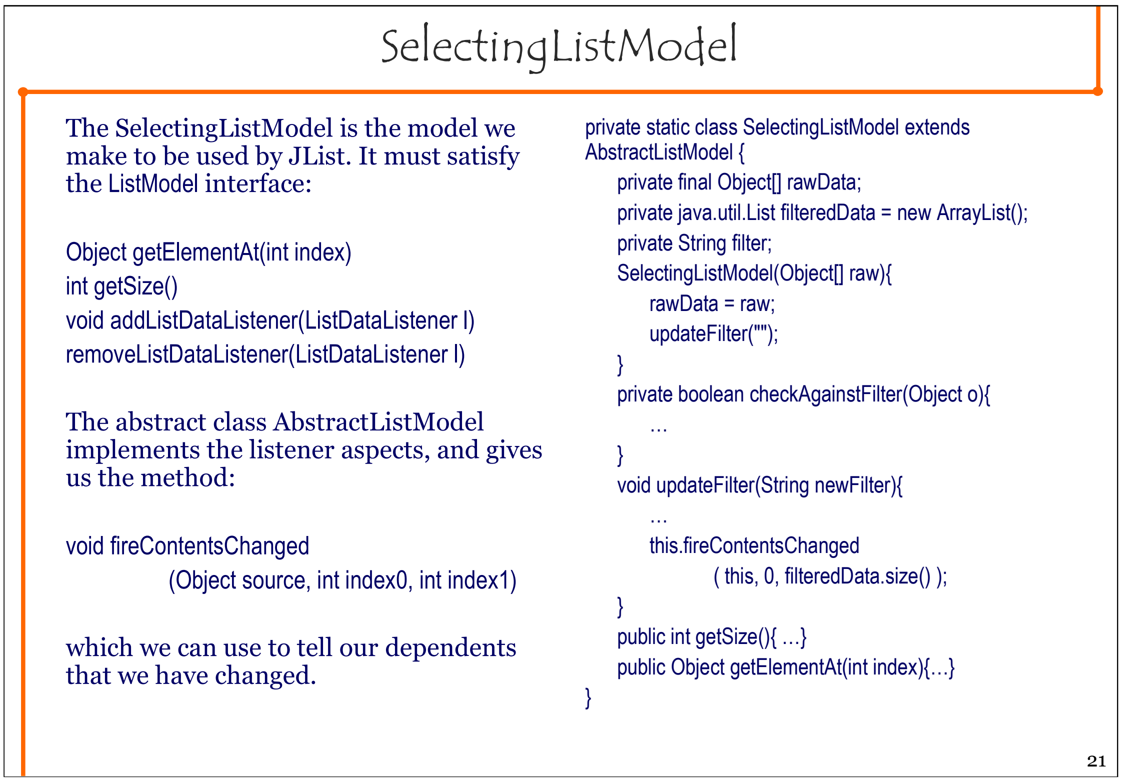
The objects behind the GUI are the SelectingList students, which internally consist of a JList jlist and a JTextField selText.

(1) the list model depends on the selText field. Changes to the text in that field imply that the list model should reduce or expand.

(2) Changes to the list model are in turn reflected in the delegate jlist (Remember JList is a delegate over the ListModel)

(3) The SelectingList students listen to changes in the selection in the selection model of jlist. When a change occurs, the students object notify about the new selection.

(4) Such changes are listened to by the OOPStudentDelegate sd. When the selection in selecting list changes, it will display the selected element (which is passed in the event object).



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){...}
}
```

21

Whenever there is an interface I, it is often a good idea to see if there is an abstract class which implements most of the interface. Here we only have to implement the getSize and getElementAt methods.

The SelectingListModel is initialized with the list of objects which is the raw (unfiltered) list.

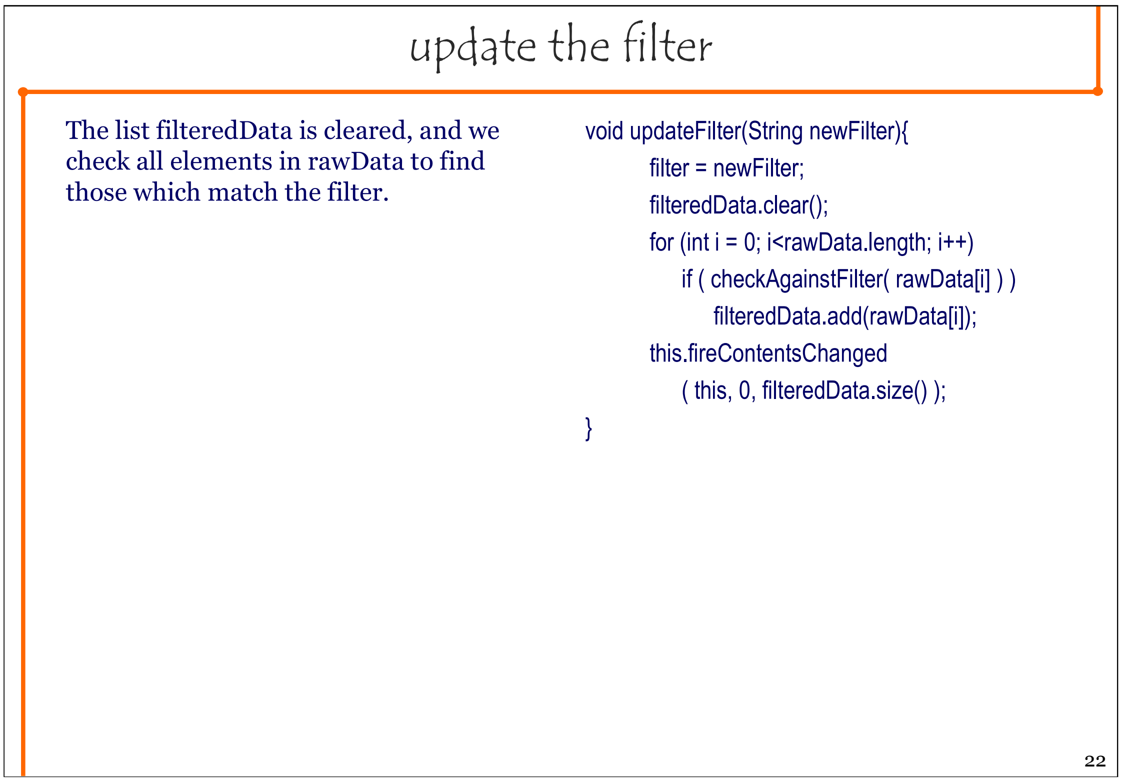
The rawData is not the data shown, instead the List filteredData is used.

The core of the SelectingListModel is the updateFilter method, which retrieves a filter method, and updates the filteredData list to contain exactly the elements in the rawData which matches the filter. When the filteredData list has been updated, we notify our dependents (which is jlist, our delegate, which in turn call getSize and getElementAt to draw a new list in the GUI).

To match the filter, an object o in the rawData, is compared to the filter string by calling the toString method on o, and seeing if the filter occurs as a substring.

```
private boolean checkAgainstFilter(Object o){
    String data = o.toString().toLowerCase();
    String filter = this.filter.toLowerCase();
    return data.indexOf(filter) >=0;
}
```

The comparison is made case insensitive.



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() );
}
```

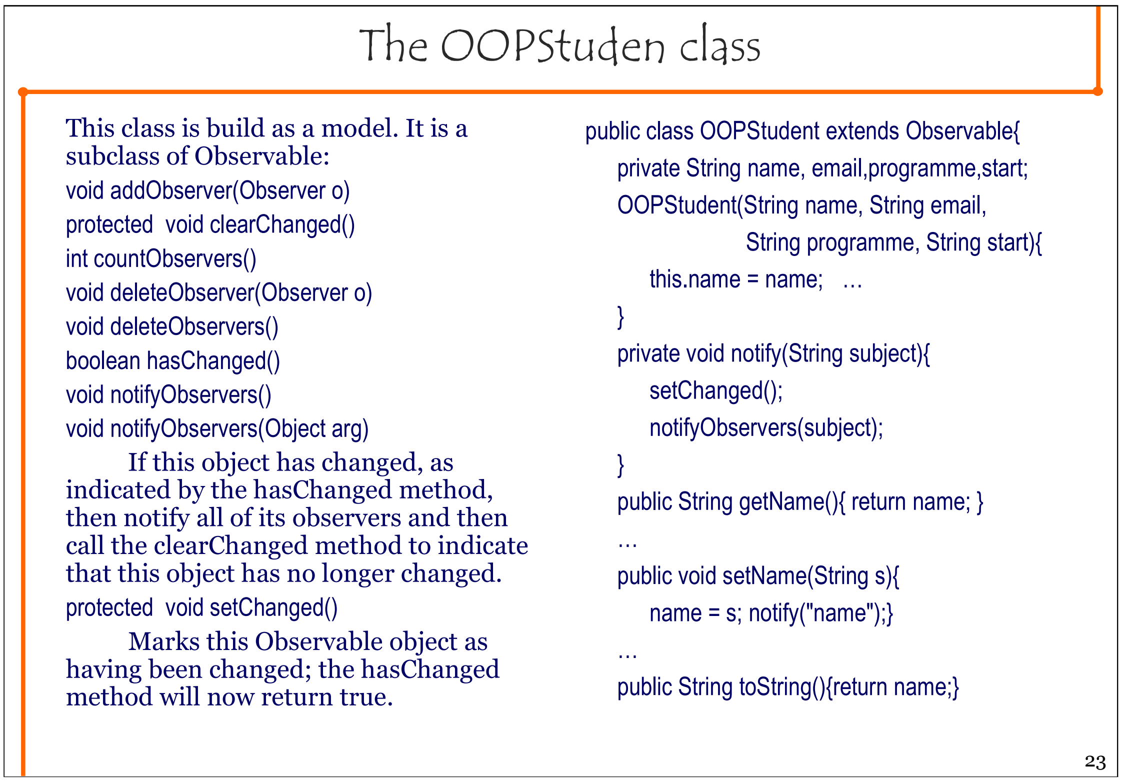
22

We do not know if the filter was extended (e.g. changed from ”Ka” to ”Kas”) or shrunk. Therefore we simply clears the filteredData, and inserts those elements that matches the new filter.

The statement

```
this.fireContentsChanged( this, 0, filteredData.size() );
```

is used to notify those listeners that depend on this list (that is, SelectingListModel) that it has changed.



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;}
```

23

Here I use the Observer interface and Observable class from java.util. I might alternatively have defined a special StudentListener interface by hand.

The class Observable has two methods which are necessary to use in the class, setChanged() and notifyObservers().

The reason is that there might be several places in the class in which changes takes place, but in which it is known that more changes are about to happen, so it is too early to notify the observers.

setChanged() registeres internally that the Observable (here OOPStudent) has changed.

notifyObservers() notifies observers of a change – if there has been a call to setChanged() before.

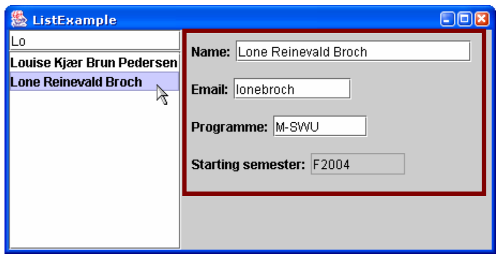
In OOPStudent, I have made a notify method, which first marks the object as changed, and then notifies all observers. One can pass an object to the observers, I pass the name of the field which was changed. That way an observer can easily check if the change was relevant.

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.



Lo

Louise Kjær Brun Pedersen

Lone Reinevald Broch

Name: Lone Reinevald Broch

Email: lonebroch

Programme: M-SWU

Starting semester: F2004

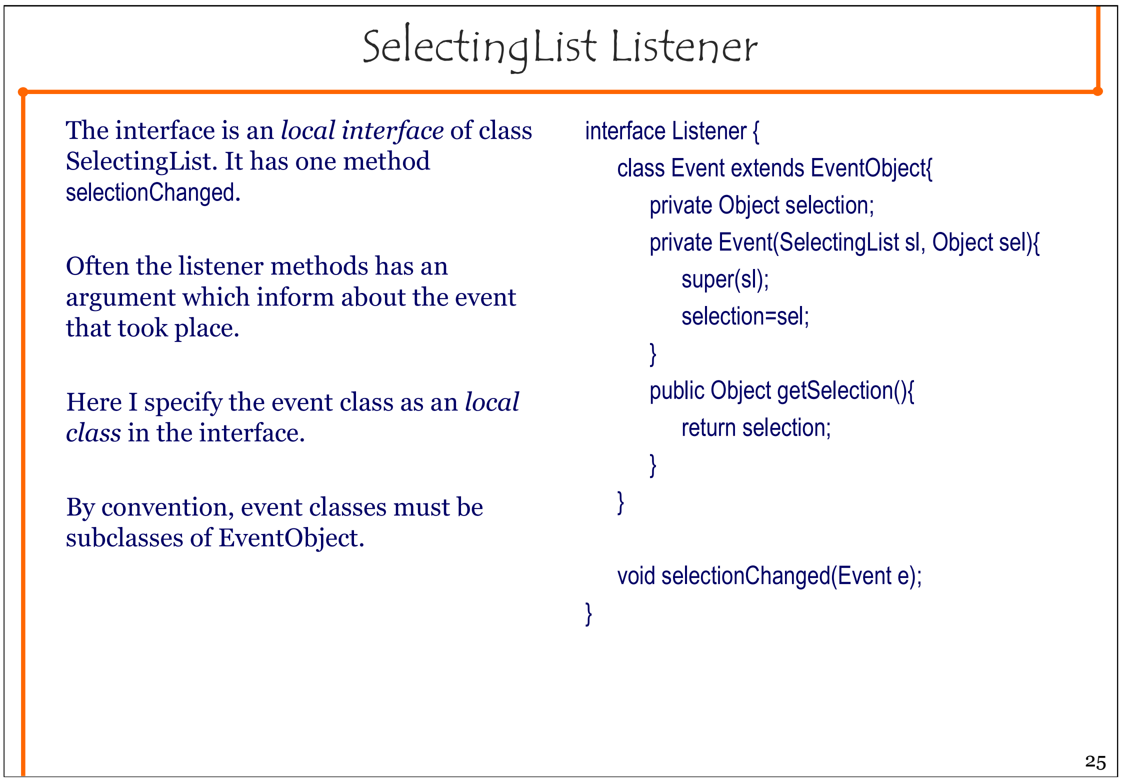
Remember, the Delegate depends on the selection in the list.

```
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);
        …
    }
}
```

24

The listener was added on slide 11. When a selection is made, the model of the delegate is changed to the selected student, which means that all fields are updated. If a de-selection occur, all fields are cleared, and made read-only.



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);
}
```

25

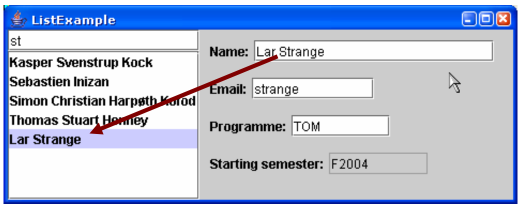
I have a tendency to use local and inner classes and interfaces whenever I can. Logically, this listener is only for the SelectingList, and the Event is only for this particular Listener.

Notice that is legal to define an local class in an interface.

The interface defines a single method selectionChanged, which takes an argument of type Event, which stores which list is was (the source of an event is stored in the superclass EventObject), and which selection was made. Null represents a de-selection.

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() );
}
```



st

Kasper Svenstrup Kock
Sebastien Inizan
Simon Christian Harpøth Korod
Thomas Stuart Henney
Lar Strange

Name: Lar Strange
Email: strange
Programme: TOM
Starting semester: F2004

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.

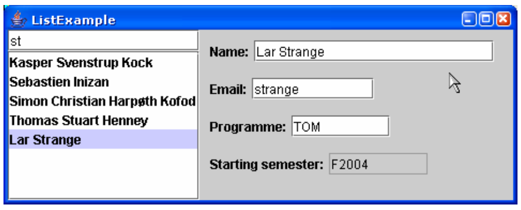
When the filter changes, we unsubscribe to all the previous elements, clear the list, and subscribe to all the new elements.

26

One could change it so that the list only listens to changes to the selected element. This requires less code and fewer listeners. The disadvantage is if there is later an extension to the program, in which one can modify students which one can change other students than the selected, then this is not reflected in the list.

Notice, if one changed the name in the example so that ”Lars Strange” became ”Lars Hansen” (Sorry Lars☺), then technically the name should disappear from the list because it no longer fulfil the filter. However, removing Lars from the list would at the same time de-select him, which in turn would imply that he was not the student shown in the right fields. I found that to be counter intuitive.

In summary



st

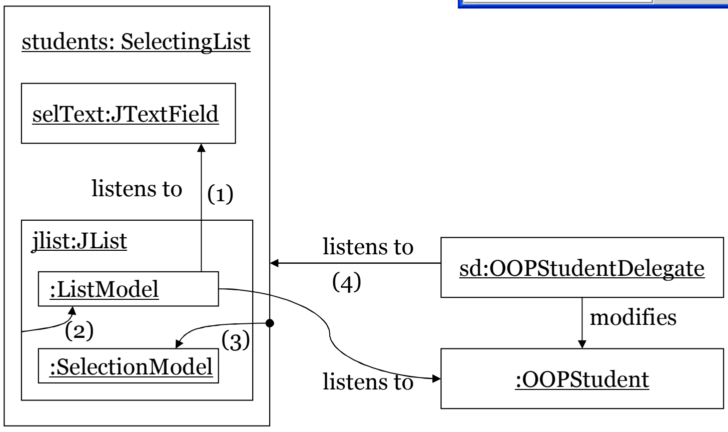
Kasper Svenstrup Kock
Sebastien Inizan
Simon Christian Harpøth Kofod
Thomas Stuart Henney
Lar Strange

Name: Lar Strange

Email: strange

Programme: TOM

Starting semester: F2004



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

28

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)

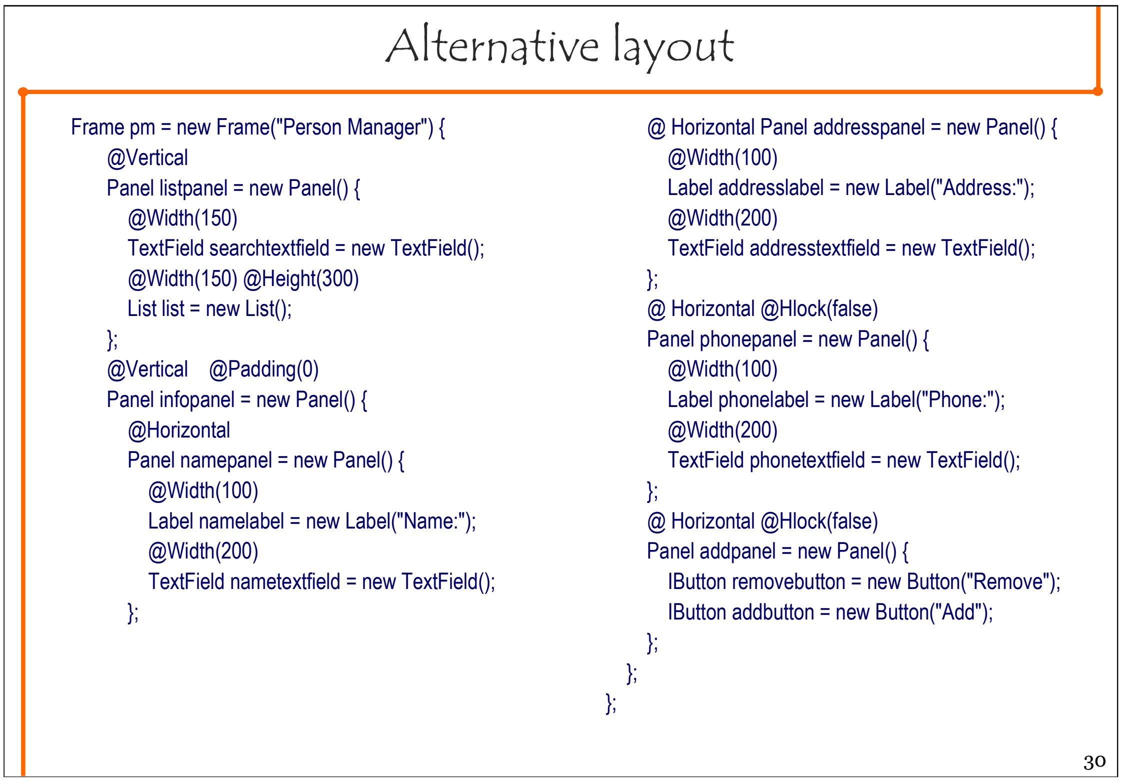
```
selText.addCaretListener(this, ”doStuff”);
…
public void doStuff(CaretEvent e){
    …
}
```

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");
        };
    };
};
```

30

This design was proposed by Thomas Quistgaard in his Masters Thesis (Speciale) Spring 2005.