

Graphical User Interfaces (GUI)

Object Oriented Programming

<http://softeng.polito.it/courses/02JEY>



SoftEng
<http://softeng.polito.it>

Version 3.4.1 - May 2016

© Marco Torchiano, Maurizio Morisio, 2016



History

- Abstract Window Toolkit (AWT)
 - Original GUI API
 - Rely on native OS components
- Java Foundation Classes (JFC)
 - Since Java 1.2
 - Includes: AWT, Swing (Widget toolkit), Java 2D
 - Lightweight
 - System independent

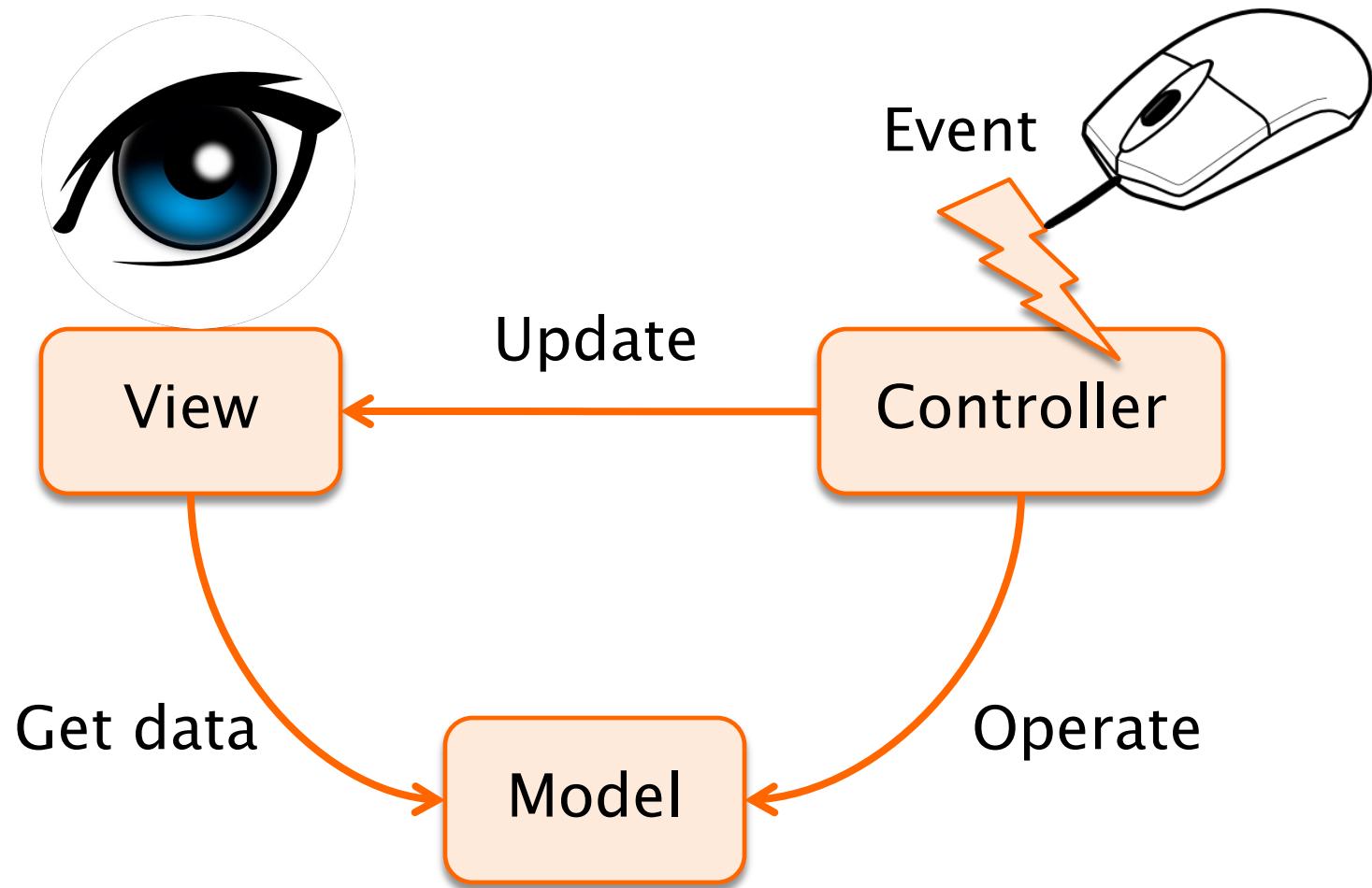
Swing

- Widget Toolkit
 - a widget (or control) is an element of a GUI that conveys information and/or represent a point of interaction
- Model–View–Controller pattern
- Pluggable look-and-feel

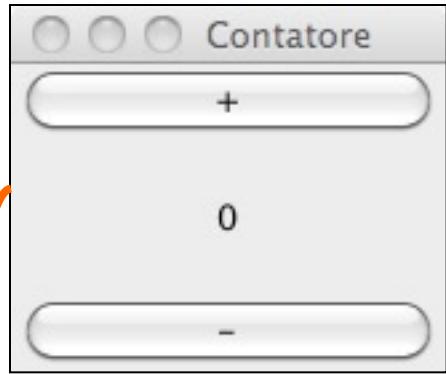


- Next technology for rich client development
 - Seamlessly integrate several different capabilities
 - FXML: markup language for UI definition
 - New graphics pipeline (Prism)
 - New Toolkit (Glass)
 - Multimedia framework
 - Web component
 - Scene Builder

MVC



MVC example



```
plusBtnClick() {  
    theCounter.increment();  
    view.update();  
}  
  
minusBtnClick() {  
    theCounter.decrement();  
    view.update();  
}
```

```
class Counter{  
    private int value;  
    public void increment() { value++; }  
    public void decrement() { value--; }  
    public int getValue() { return value; }  
}
```

MVC Principles

- When building a GUI we must consider two main aspects:
 - Layout (View): how to place the graphical elements to achieve a give visual aspect
 - Events (Controller): which behavior associate to elements' events
- Application logic (Model) must remain, as far as possible, separate from user interface.

Execution flow

- There is no predefined order of execution in GUI applications
 - Operations are performed in response to external events (e.g. mouse click)
 - Event handling is serialized
 - To execute several operations in parallel, threads must be used
- Method `main` in GUIs has the only goal of instantiating the graphical elements

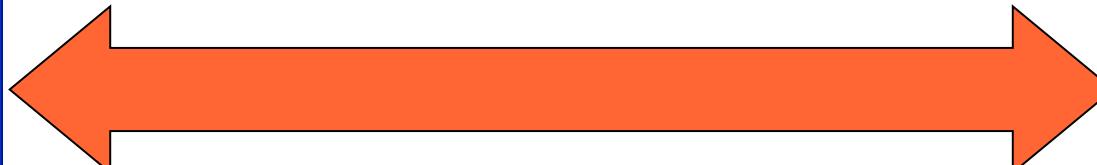
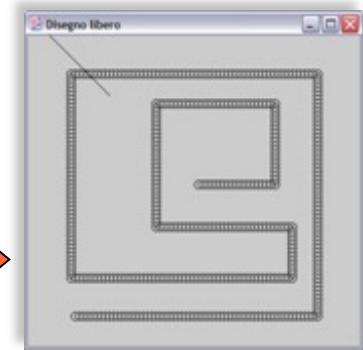
GUI categories

- It is possible to identify two extreme types of GUIs:
 - Component composition
 - Direct drawing
- In practice a mix of the two is used.



Components

Direct drawing



GUI with Components

- Use predefined UI components (widgets or controls)
 - E.g. buttons, text fields, labels
- They manage mostly textual information
 - Suitable to build an application that could “theoretically” make it with a textual user interface

GUI with drawing

- They directly access the screen
 - The tool is represented by the **Graphics** interface
 - They may use sophisticate API such as Java2D
- They manage visual informations (e.g. diagrams, graphs, images)
- Typically are contained in a JPanel component

MAIN CONTAINER

Main GUI container

- Represents the point of interaction between Java and Operating System (OS)
- It may vary:
 - JFrame for desktop applications
 - Applet for web-enabled components
 - Midlet for JavaME (phone) applications
 - ...

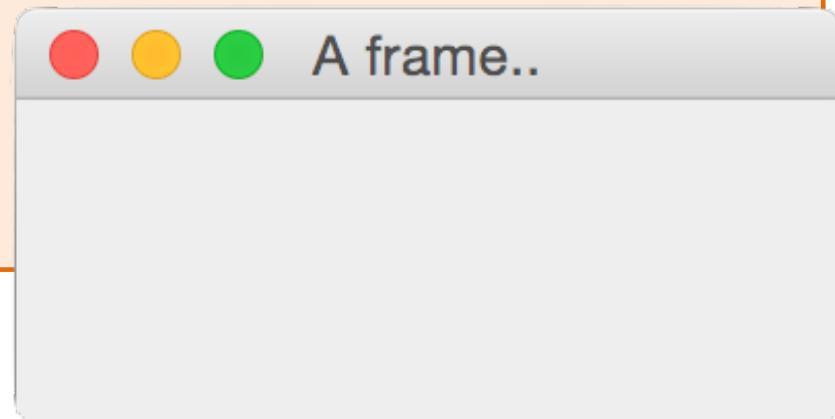
Frame container

- **JFrame** is the base class for desktop graphical applications
- Provides all the features for an empty window
 - Title bar
 - Standard buttons (Max, Min, Close)
 - Resizable border
 - Etc.

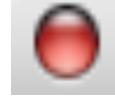
JFrame – Example

```
public class BasicFrame extends JFrame {  
    public static void main(String[] args) {  
        JFrame f = new BasicFrame();  
        f.setVisible(true);  
    }  
    public BasicFrame() {  
        super("A frame..");  
        setSize(200,100);  
    }  
}
```

What happens
without setSize() ?



Window close

- Clicking the button  or 
 - Closes the window but
 - **DO NOT** terminate the application
- It is required to **explicitly** define the operation to be performed in response to window closure

E.g.

```
setDefaultCloseOperation(  
    JFrame.DISPOSE_ON_CLOSE);
```

Container Basic Features

- `setDefaultCloseOperation(bhvr)`: define the behavior upon window close
 - `EXIT_ON_CLOSE`
 - `DO NOTHING ON CLOSE`
 - `DISPOSE ON CLOSE`
 - `HIDE ON CLOSE`
- `setSize(int width, int height)`: defines the dimensions of the panel outside

Application

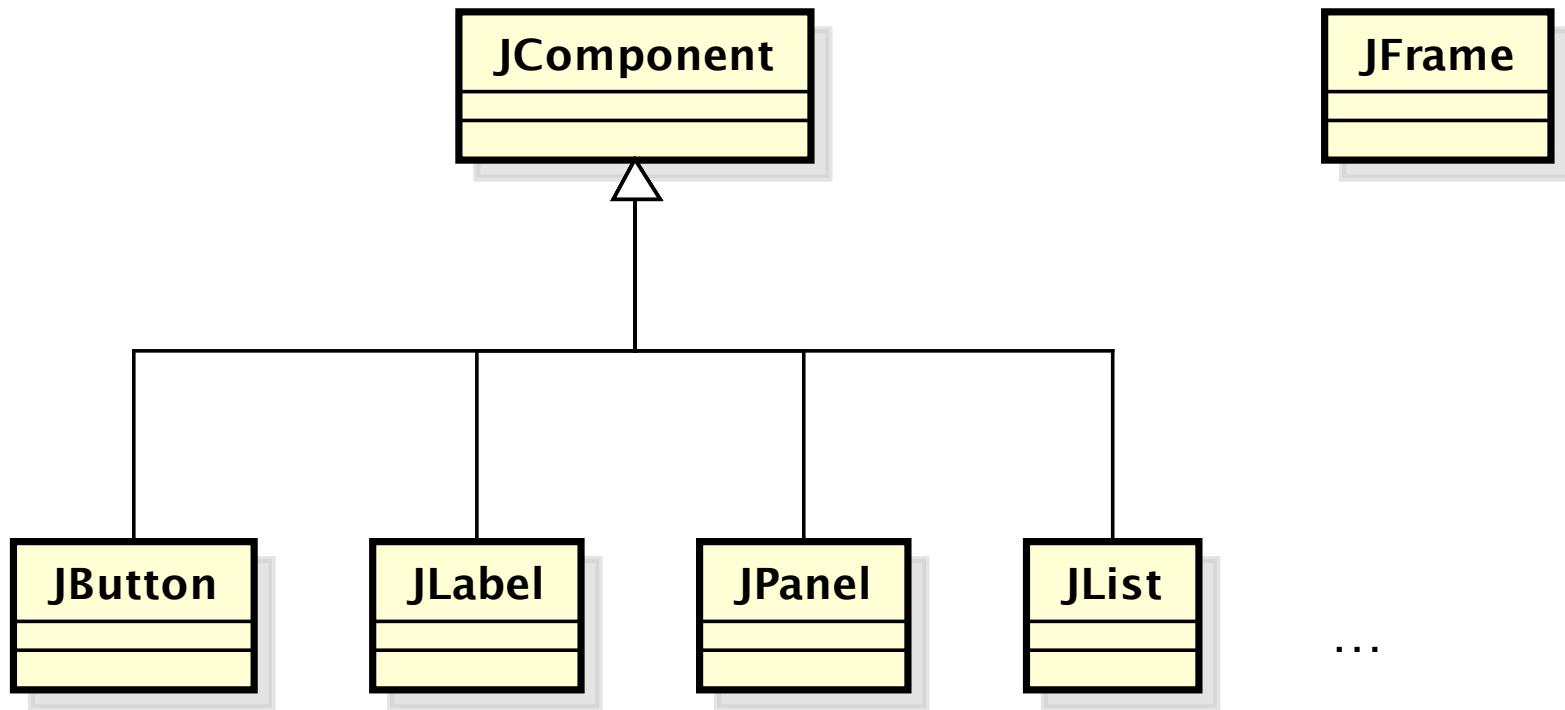
- The main application has to
 - create a container (**JFrame**)
 - make it visible (launches a new event thread)

```
public class Minimal extends JFrame {  
    public Minimal() {  
        setTitle("Minimal UI");  
        setSize(200,100);  
        setDefaultCloseOperation(EXIT_ON_CLOSE);  
    }  
    public static void main(String[] args) {  
        Minimal gui = new Minimal();  
        gui.setVisible(true);  
    } }
```

Could be moved
into the ctor

COMPONENTS

Main classes



Button: JButton

- Constructors:
 - **JButton()** ; creates a button without a text (without label)
 - **JButton(String)** ; creates a button with a label containing the text.
- it is a component → inherits all the methods of classes JComponent (javax.swing) and component (java.awt)
- It is a container → inherits all methods of java.awt.Container

Label: JLabel

- Constructors

- `JLabel(String)`; create label with given text, aligned on the left
- `JLabel(String, int)`; create label with given text, aligned:
 - `SwingConstants.LEFT`
 - `SwingConstants.RIGHT`
 - `SwingConstants.CENTER`

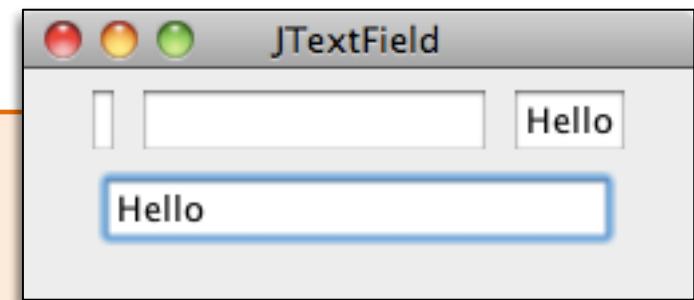
- Available methods:

- `getText()`, `setText(String)`
- `getAlignment()`, `setAlignment(int)`

Text field: JTextField

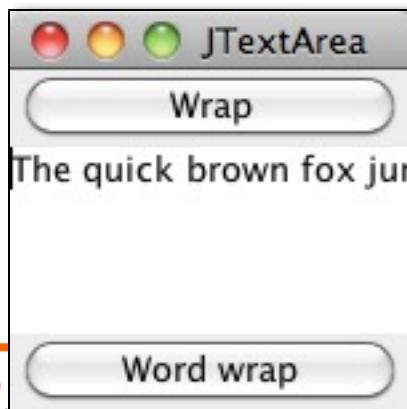
- The text fields allows entering strings of text on a single line
- Constructors:
 - **JTextField(String)** initial content
 - **JTextField(int)** required size in chars
 - ...

```
add(new JTextField());  
add(new JTextField("", 20));  
add(new JTextField("Hello"));  
add(new JTextField("Hello", 30));
```



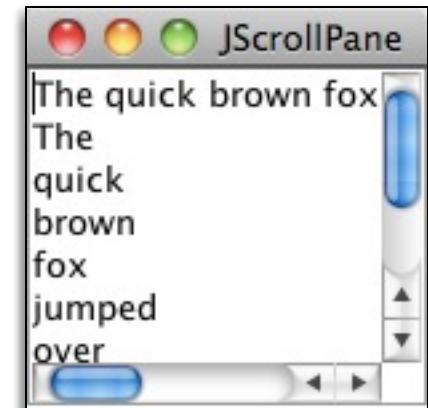
Text area: JTextArea

- Manages text on several lines
- Constructors
 - `JTextArea (int lines, int columns)`
 - `JTextArea (String text, int l, int c)`
- Useful Methods:
 - `getText()`, `setText(String)`;
 - `append(String)`, `insert(String, int)` ;
 - `void setLineWrap(boolean)`
 - `void setWrapStyleWord(boolean)`



ScrollPane: JScrollPane

- **JScrollPane** is able to add scroll bars to a scrollable component (e.g. **JTextArea**)
- Constructor:
 - **JScrollPane(Component)** ;
- Example:
 - **JScrollPane sp =
new JScrollPane(
 new JTextArea(longText));**

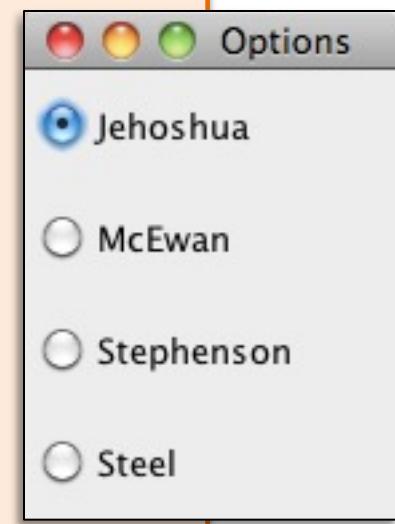


Checkbox, Options

- Check boxes : **JCheckBox (String, boolean)**
- Option buttons: **JRadioButton (String, boolean)**
- Useful methods:
 - `void setSelected(boolean)`
 - `boolean isSelected()`
- Mutual exclusion:
 - Add RadioButton (or CheckBox) to **ButtonGroup**
 - By default they are non-exclusive

Example

```
public class Authors extends JFrame{
JRadioButton[] list = new JRadioButton[4];
public Authors() {
    super("Select an author");
    setSize(140, 190);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    list[0] = new JRadioButton("Jehoshua", true);
    list[1] = new JRadioButton("McEwan");
    list[2] = new JRadioButton("Stephenson");
    list[3] = new JRadioButton("Steel");
    JPanel panel = new JPanel();
    ButtonGroup group = new ButtonGroup();
    for (int i = 0; i < list.length; i++) {
        group.add(list[i]);
        panel.add(list[i]);
    }
    setContentPane(panel);
    setVisible(true);
}
public static void main (String args[]) {
    Authors newLista = new Authors(); }
```



Dialog boxes

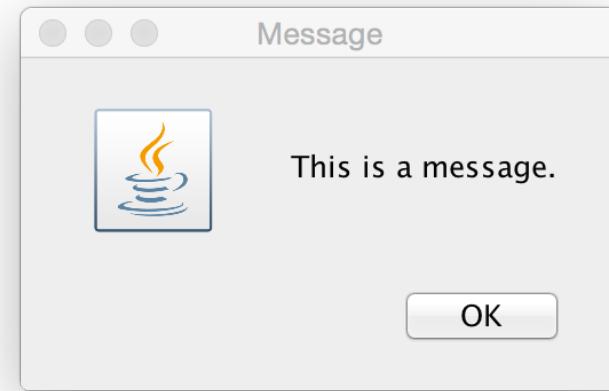
- Used for short focused interactions
 - Confirmation
 - Input
 - Message
 - Options
- Methods more efficient than input/output in order to read from keyboard
- Class **JOptionPane**
 - Several static methods for different types

Dialog for confirmation

- Every dialog is dependent on a root Frame component.

- Example:

```
JOptionPane.showMessageDialog(frame,  
                            "This is a message.");
```



JOptionPane Features

Using **JOptionPane**, you can quickly create and customize several different kinds of dialogs.

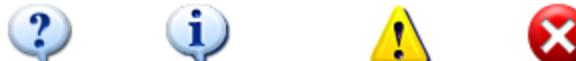
JOptionPane provides support for laying out standard dialogs, providing icons, specifying the dialog title and text, and customizing the button text.

Icons used by JOptionPane
(Java look and feel)



question information warning error

(Windows look and feel)

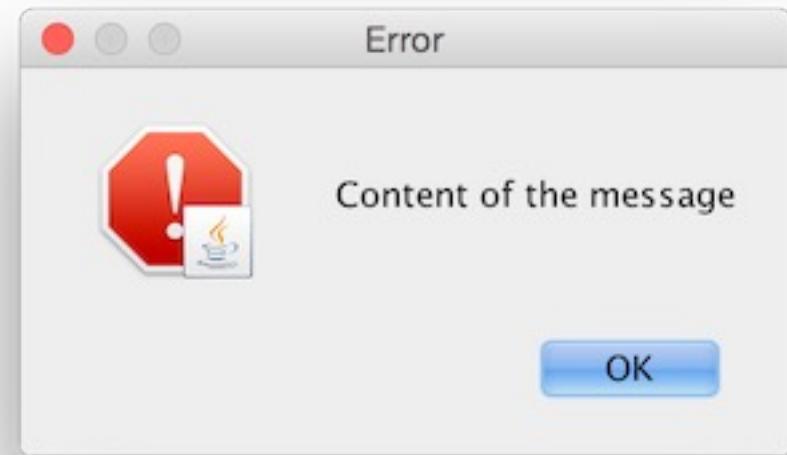


question information warning error

Message dialog types



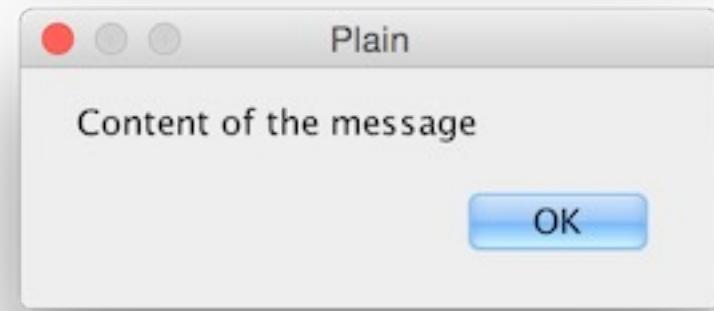
INFORMATION_MESSAGE (default)



ERROR_MESSAGE



WARNING_MESSAGE

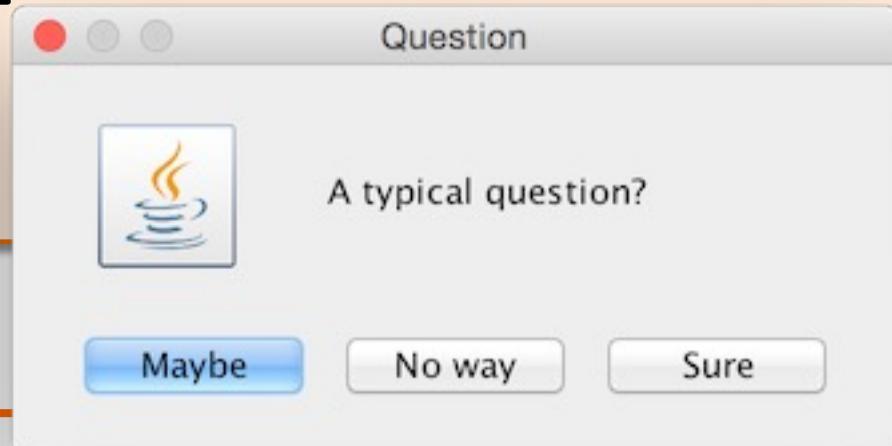


PLAIN_MESSAGE

Option dialog

- Presents used with a few choices

```
String[] options={"Sure","No way","Maybe"};  
JOptionPane.showOptionDialog(null,  
    "A typical question?",  
    "Question",  
    JOptionPane.YES_NO_CANCEL_OPTION,  
    JOptionPane.QUESTION_MESSAGE,null,  
    Options,  
    options[1]);
```

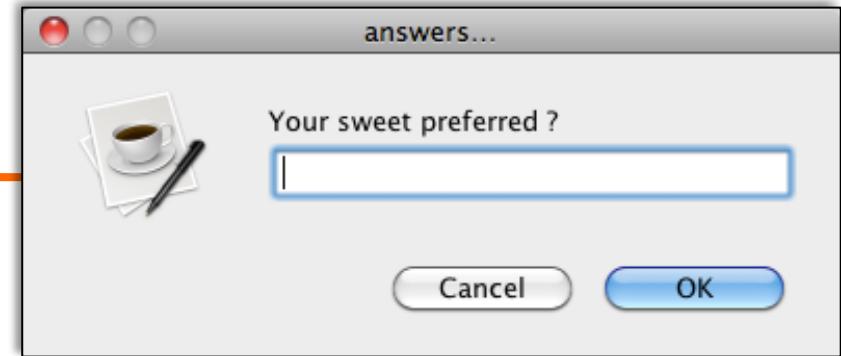


Input dialog

- Methods

- `String showInputDialog(Component, Object)`
- `String showInputDialog(Component, Object, String, int)`
 - Component: in which component appears window
 - Object: Request message input
 - String: title
 - int: type of message (as in confirmation)

```
String answer =  
    JOptionPane.showInputDialog(null,  
    "Your sweet preferred ?", "answers... ",  
    JOptionPane.QUESTION_MESSAGE);
```



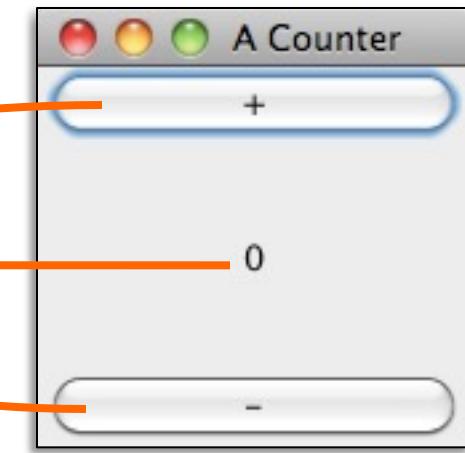
Full example: visual counter

- Model: a simple counter

```
public class Counter {  
    private int value;  
    public void increment() {  
        value++;  
    }  
    public void decrement() {  
        value--;  
    }  
    public int getValue() {  
        return value;  
    }  
}
```

Visual Counter – View

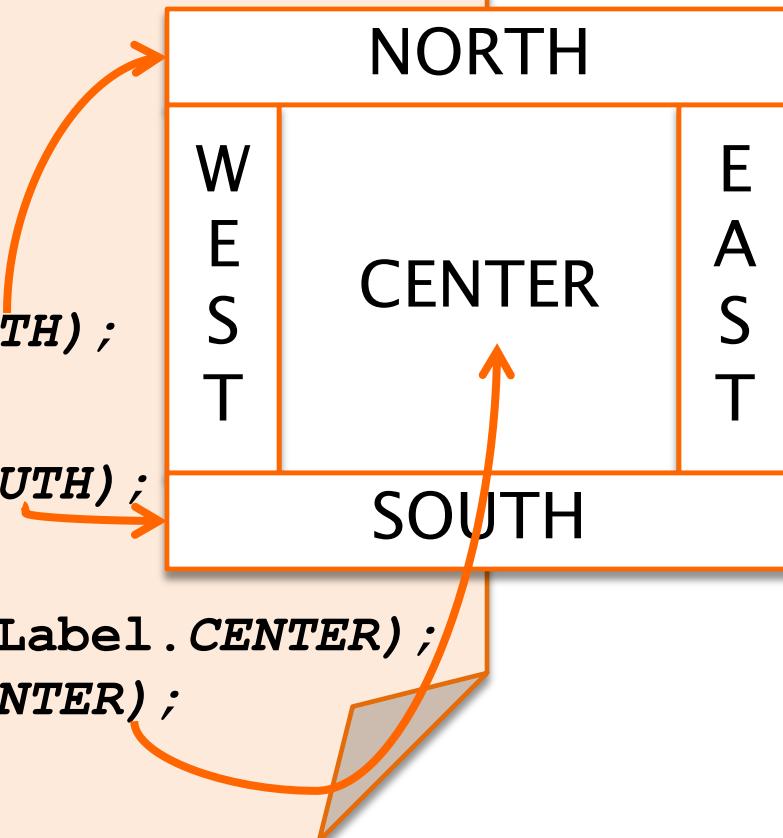
```
public class View extends JFrame {  
    private JButton plus; ←  
    private JLabel value; ←  
    private JButton minus; ←  
    private Counter model; →  
    public View(Counter c,  
               Controller controller) {  
        ... }  
    public void update() {  
        ... }  
}
```



Model

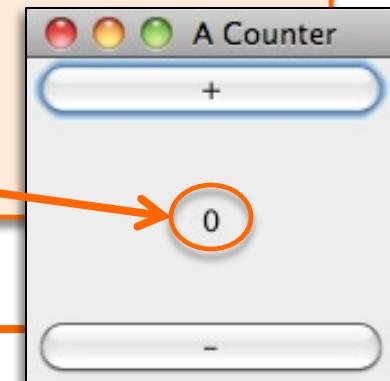
Visual Counter – View

```
public class View extends JFrame {  
  
    public View(Counter c, Controller controller) {  
  
        setTitle("A Counter");  
        setSize(150,150);  
        setLayout(new BorderLayout());  
        plus = new JButton("+");  
        this.add(plus, BorderLayout.NORTH);  
  
        minus = new JButton("-");  
        this.add(minus, BorderLayout.SOUTH);  
  
        value = new JLabel(?);  
        value.setHorizontalAlignment(JLabel.CENTER);  
        this.add(value, BorderLayout.CENTER);  
  
        setVisible(true);  
    }  
}
```



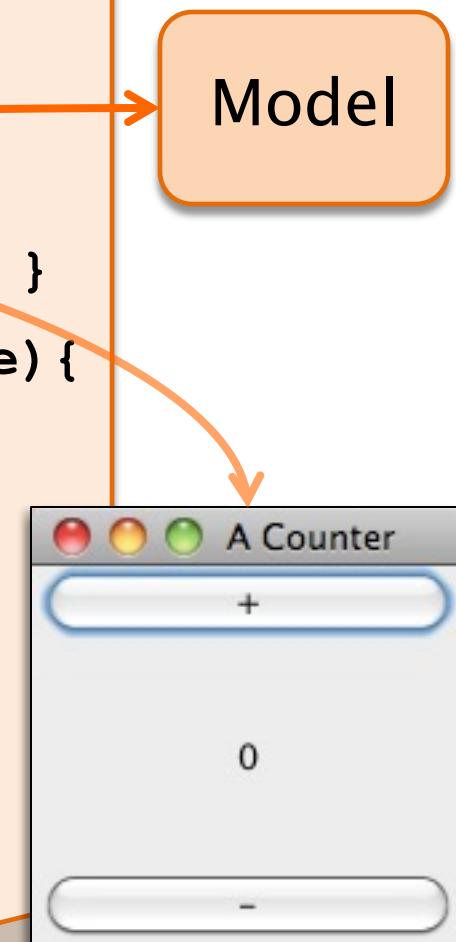
Visual Counter – View

```
setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);  
  
model = c; // MODEL  
  
plus.addActionListener(controller);  
minus.addActionListener(controller);  
  
controller.setView(this); // CONTROLLER --> VIEW  
  
update();  
}  
  
public void update() {  
    String v = Integer.toString(model.getValue());  
    value.setText(v);  
}
```



Visual Counter – Controller

```
public class Controller
    implements ActionListener {
private Counter model;
private View view;
public Controller(Counter m) { model = m; }
public void actionPerformed(ActionEvent e) {
    if(e.getActionCommand().equals("+")) {
        model.increment();
    } else
        model.decrement();
    view.update();
}
public void setView(View finestra) {
    view = finestra;
}
}
```



LAYOUT

Layout managers

- Determine the size and position of the components within a container
 - Manage resize of containers
 - Accounts for differences in OSs and font sizes
- **setLayout (LayoutManager m) ;**
- Absolute positioning is possible
 - **setLayout(null) ;**
 - **setBounds ()** for each component

Flow Layout

- From left to right, starting from the left upper corner
- Constructors:
 - `FlowLayout()`;
 - `FlowLayout(int align)`;
 - `FlowLayout(int align, int hgap, int vgap)`;
- Parameters:
 - `align`: Alignment of basis (`FlowLayout.LEFT`,
`FlowLayout.RIGHT`, `FlowLayout.CENTER`)
 - `hgap`: Horizontal space between components (default: 3 px)
 - `vgap`: Vertical space between components (default: 3 px)

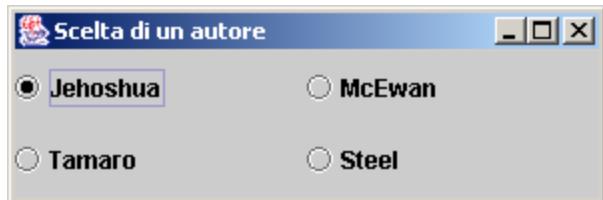
Grid Layout

- Splits the screen in a grid of rows and columns
- Filling: starts from the box in the top left and then by line
- Constructors:
 - `GridLayout(int rows, int cols)`
 - `GridLayout(int rows,int cols,int hgap,int vgap)`
- Parameters:
 - `rows`: number of row;
 - `cols`: number of columns;
 - `hgap`: Spacing (in pixels) between two horizontal boxes
(default: 0 pixel)
 - `vgap`: spacing (in pixel) between two vertical boxes
(default: 0 pixel)

Example of GridLayout



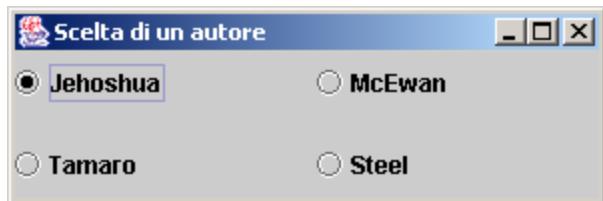
4 row, 1 columns:
distance 0



2 row, 2 columns:
distance 0 pixel



(Distanza min = 0)



2 row, 2 columns:
distance 10 pixel

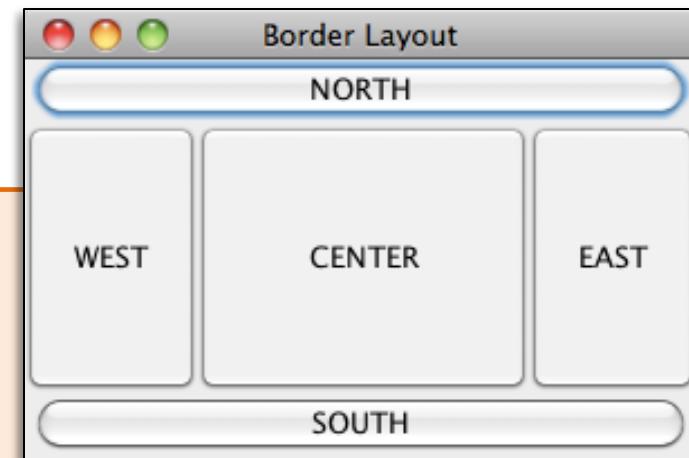


(Min distance = 10)

BorderLayout

- Divide the container into five areas
 - 4 in the border 1 in the center

```
setLayout(new BorderLayout());  
add("North",new JButton("NORTH"));  
add("South",new JButton("SOUTH"));  
add("East",new JButton("EAST"));  
add(new JButton("WEST"),BorderLayout.WEST);  
add(new JButton("CENTER"),BorderLayout.CENTER);
```



Grid bag layout

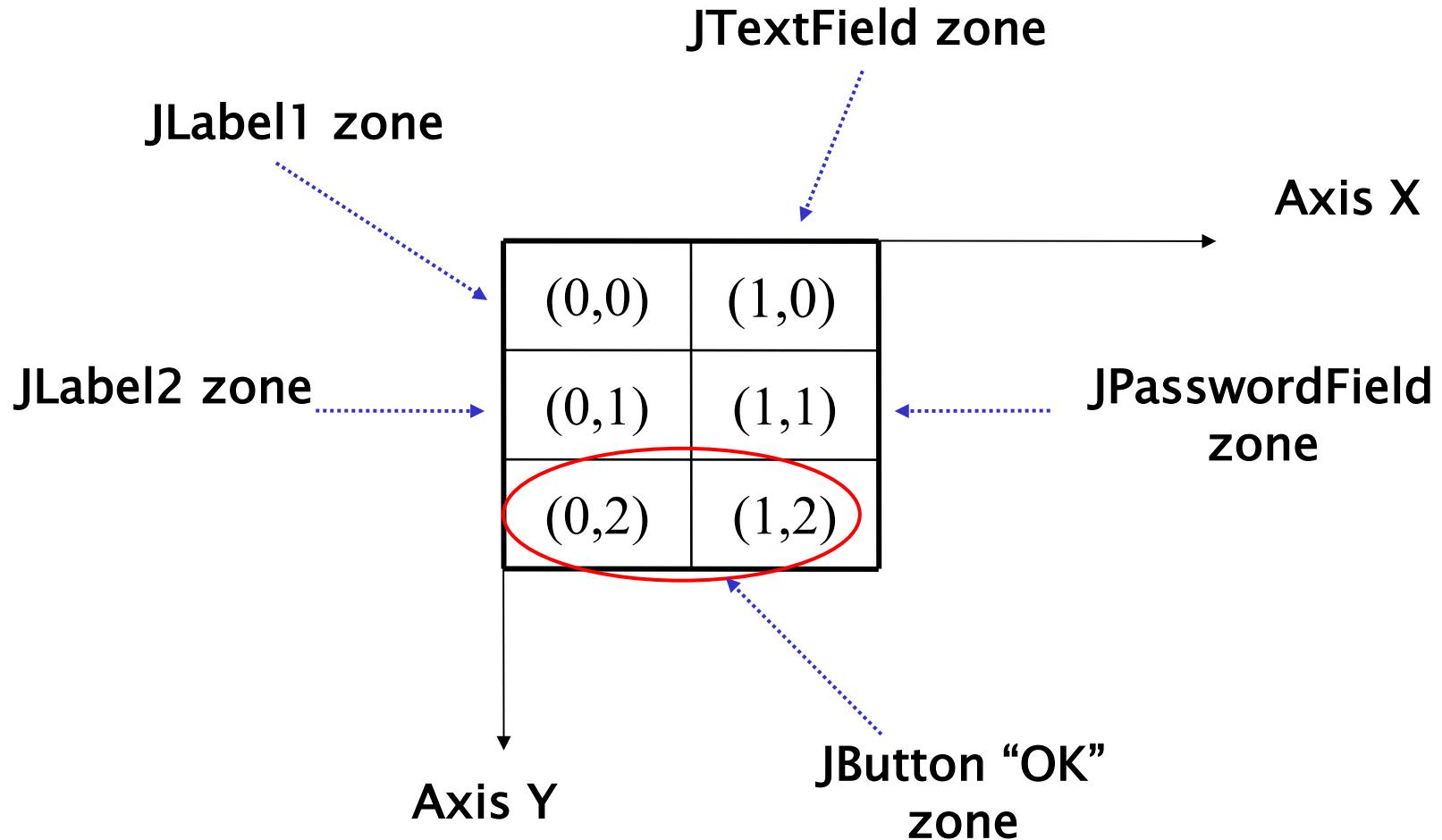
- Extension of the grid layout (GridLayout)
 - elements of the grid can be adjusted with mechanisms of personalization
- Usage procedure:
 - Create **GridBagLayout** object
 - Create ‘constraint’ object (**GridBagConstraints**)
 - For each component
 - Define the adjustment
 - Register the component–constraint link with the manager
 - Add the component to the container

Example

- What is the structure required to have a simple login window as:



GridBagConstraints details



Rules expressed as constraints

- Components are placed in the cells at position (x, y)
- "OK" button must occupy two cells: the other components are in a single cell
- breadth of the components is variable (the label "name" occupies about 30% of line...)
- Cells are positioned (the "OK" button is centered, etc.)

Rules on GridBagConstraints (2)

- GridBagConstraints has different fields:
 - **gridx** – The initial gridx value.
 - **gridy** – The initial gridy value.
 - **gridwidth** – The initial gridwidth value.
 - **gridheight** – The initial gridheight value.
 - **weightx** – The initial weightx value.
 - **weighty** – The initial weighty value.
 - **anchor** – The initial anchor value.
 - **fill** – The initial fill value.
 - **insets** – The initial insets value.
 - **ipadx** – The initial ipadx value.
 - **ipady** – The initial ipady value.

Regolation on GridBagConstraints (3)

- The values of **fill** are : **BOTH**, **NONE**, **HORIZONTAL**, **VERTICAL**
- The values of **anchor** are: **CENTER**, **NORTH**, **NORTHEAST**, **EAST**, **SOUTHEAST**, **SOUTH**, **SOUTHWEST**, **WEST**, **NORTHWEST**
- Therefore...

```
GridBagLayout grid = new GridBagLayout();
pannel.setLayout(grid);
GridBagConstraints Gbc = new GridBagConstraints();
JLabel label1 = new JLabel ("Name:", JLabel.LEFT);
Gbc.gridx = 0;
Gbc.gridy = 0;
Gbc.gridwidth = 1;
Gbc.gridheight = 1;
Gbc.weightx = 30;
Gbc.weighty = 40;
Gbc.fill = GridBagConstraints.NONE;
Gbc.anchor = GridBagConstraints.EAST;
grid.setConstraints(Gbc, label1);
pannello.add(label1);
```

Java Events



SoftEng
<http://softeng.polito.it>

Event Delegation Model

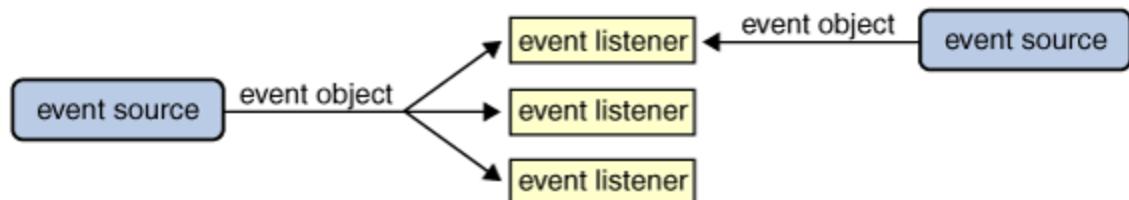
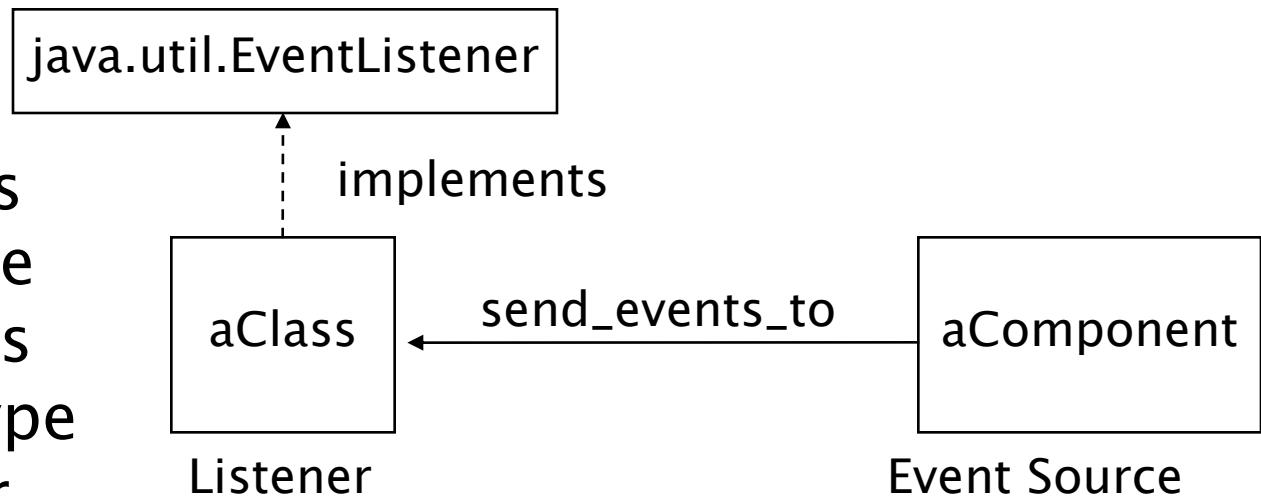
- Since Java 1.1
 - Events are classified by type (MouseEvent, KeyEvent, etc.)
 - Events are generated in components sources
 - An object can be registered as handler (listener) of a type of event by sending a message to the component source

Event Delegation Model

- Whenever an event occurs, the AWT thread sends a message to all the registered listener objects (the event descriptor is passed as a parameter)
- A listener object must implement the appropriate interface (to make possible the call-back)

Event Delegation Model

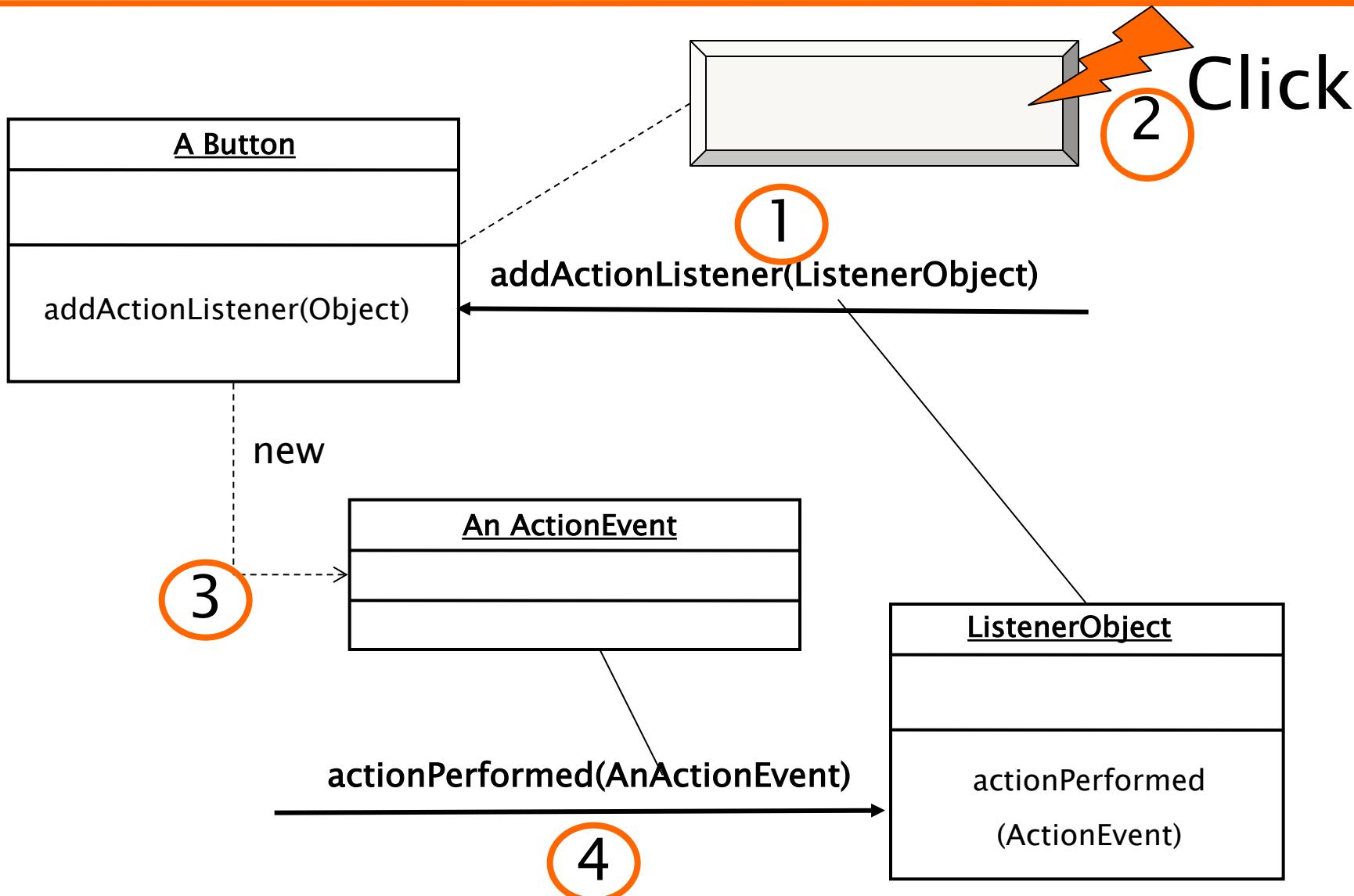
Multiple listeners can register to be notified of events of a particular type from a particular source. Also, the same listener can listen to notifications from different objects.



Events

- The events are represented by a hierarchy of classes. Each class is defined by the data representing that type of event.
- Some of the classes that are a set of events (MouseEvent) MAY CONTAIN AN ID that identifies the exact event type.

Example events



Management of the events

- Events covered in Java :
 - **Action event** → click a button
 - **Adjustment event** → actions on scroll bars
 - **Focus event** → point the mouse on a text field
 - **Item event** → click on RadioButton, CheckBoxButton
 - **Key event** → keyboard input
 - **Mouse event** → click (not covered above)
 - **Mouse-motion event** → Simple displacement of the mouse
 - **Window event** → Enlarge, close a window

Managing events

- The principle underlying the events is quite similar to the exceptions :
 - A class declares which event is able to deal with (one or more) → implements one or more interfaces
 - It joins the listener set of the components that are source of events (`JButton`, `JTextField`, etc..) →
`aButton.addActionListener(controller)`
 - Pay attention! You're implementing interfaces, so you must overwrite all methods of those interfaces!

How to manage the events in Java

```
class FrameWithEvents extends JFrame implements  
InterfaceWithEvents {  
  
    JComponent componentSourceofEvents =  
        new JComponent();  
  
    componentSourceOfEvents.addListener(this);  
  
    void methodOfTheInterfaceWithEvents() {...}  
    void anotehrMethodOfTheInterfaceWithEvents() {...}  
}  
//end class
```

Listener Interfaces (1)

- **ActionListener** → Methods to override
 - :
 - :
 - `void actionPerformed (ActionEvent evt)`
- **FocusListener** → Methods to overwrite
 - :
 - :
 - `void focusGained (FocusEvent evt)`
 - `void focusLost (FocusEvent evt)`
- **ItemListener** → Methods to rewrite :
 - `void itemStateChanged (ItemEvent e)`

Listener Interfaces (2)

- **MouseListener** Methods to override:
 - `void mouseClicked (MouseEvent evt)`
 - `void mouseEntered (MouseEvent evt)`
 - `void mouseExited (MouseEvent evt)`
 - `void mousePressed (MouseEvent evt)`
 - `void mouseReleased (MouseEvent evt)`
- **MouseMotionListener** Methods to overrid:
 - `void mouseDragged (MouseEvent evt)`
 - `void mouseMoved (MouseEvent evt)`

Listener Interfaces (3)

- **KeyListener** Methods to override:
 - `void keyPressed(KeyEvent evt)`
 - `void keyReleased(KeyEvent evt)`
 - `void keyTyped(KeyEvent evt)`
- **WindowListener** (Methods to override:
 - `void windowActivated(WindowEvent evt)`
 - `void windowClosed (WindowEvent evt)`
 - `void windowClosing (WindowEvent evt)`
 - `void windowDeactivated (WindowEvent evt)`
 - `void windowDeiconified (WindowEvent evt)`
 - `void windowIconified (WindowEvent evt)`
 - `void windowOpened (WindowEvent evt)`

Add a listener

- Separate controller object

```
button.addActionListener( controller );
```

- Lambda expression relaying call

```
button.addActionListener( e -> doClick() );
```

- The container itself (e.g. JFrame)

```
button.addActionListener( this );
```

Handle the event

- Identify the source of events
 - May be implicit in the anonym dispatcher

```
Object ob = evt.getSource();  
if (ob == button) {  
    // perform event handling  
}
```

Note: ==
reference
comparison

- Use event additional information
 - E.g. mouse position

Handle the event

- All methods accept an event as argument
 - The argument (KeyEvent, MouseEvent, etc.) provides methods to get information about the event:
- Examples
 - ActionListener
 - `String getActionCommand()`: returns a string identifying the component which generated the command
 - `String paramString()`: returns a string describing the event type (common to all event objects)

Event methods

- **ItemEvent:**
 - int **getStateChange()**: return SELECTED or DESELECTED on whether the RadioButton or the CheckBox is turned on or off
- **KeyEvent:**
 - char **getKeyChar()**: returns the character typed
 - int **getKeyCode()**: returns the code of the key pressed or released

GUI TESTING

GUI testing

- To execute a test of a GUI there are two possible approaches:
 - Test from outside
 - Test from within

Test from outside

- Test from outside
 - Through the Operating System events are sent to the application emulating the user behavior
 - Pro: realistic approach
 - Cont: complex, OS dependent
 - There are specific tools that are able to capture operations performed by a user and to replay them later

Test from within

- Test from within
 - Specific methods can be invoked on graphical component to achieve a similar effect to that of a real usage (e.g. doClick() on a button)
 - Pro: simple, OS independent
 - Con: not realistic, not full interaction
 - Con: classes must be designed for testability
 - E.g. let selected attribute visible

GUI Test – Example

```
public void testGUI() {  
    SimpleCassa gui = new SimpleCassa();  
    gui.setVisible(true);  
  
    gui.input.setText("P001");  
    gui.pulsante.doClick();  
    gui.input.setText("P002");  
    gui.pulsante.doClick();  
    gui.input.setText("P001");  
    gui.pulsante.doClick();  
  
    gui.input.setText("CLOSE");  
    gui.pulsante.doClick();  
  
    String output = gui.output.getText();  
    assertTrue("wrong output",  
              output.indexOf("Sum: 7.5") > -1);  
}
```

GRAPHICS

Direct drawing

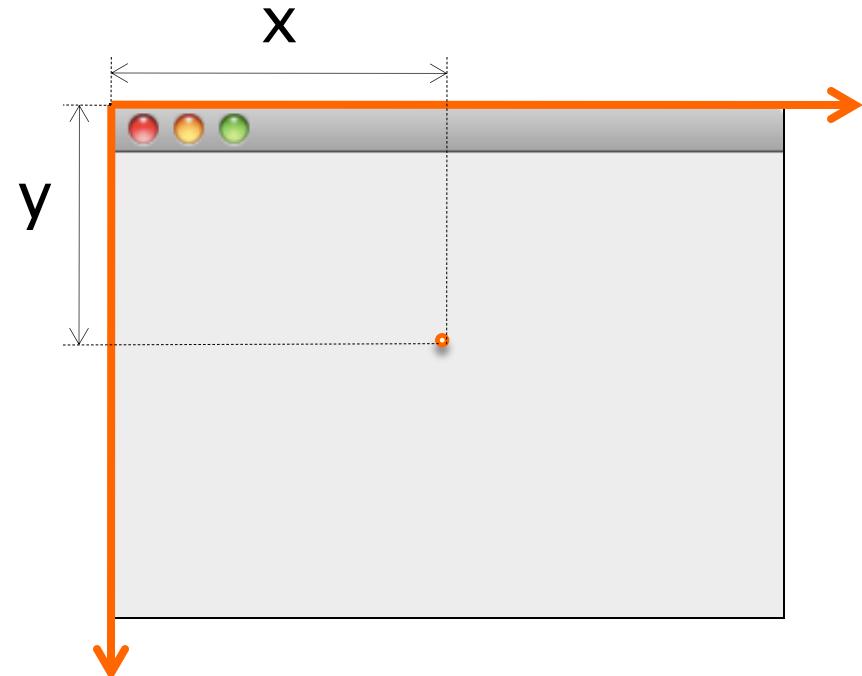
- Two elements are required to draw directly :
 - Method **void paint(Graphics g)**
 - Must be redefined in derived classes
 - Invoked by O.S.
 - Class **Graphics**
 - Provides methods to draw

Class Graphics

- Is the class that supports the capacity graphics applications, which draw lines, forms, characters and present images on screen, by means of a series of methods .
- The method **paint()** provides an object graphics acting on which draws on the screen.
 - It isn't necessary to create an instance of the class graphics to draw on the screen

Class Graphics

- The coordinate system:
 - Origin in the top left corner
 - X increase moving to the right
 - Y increase moving downwards

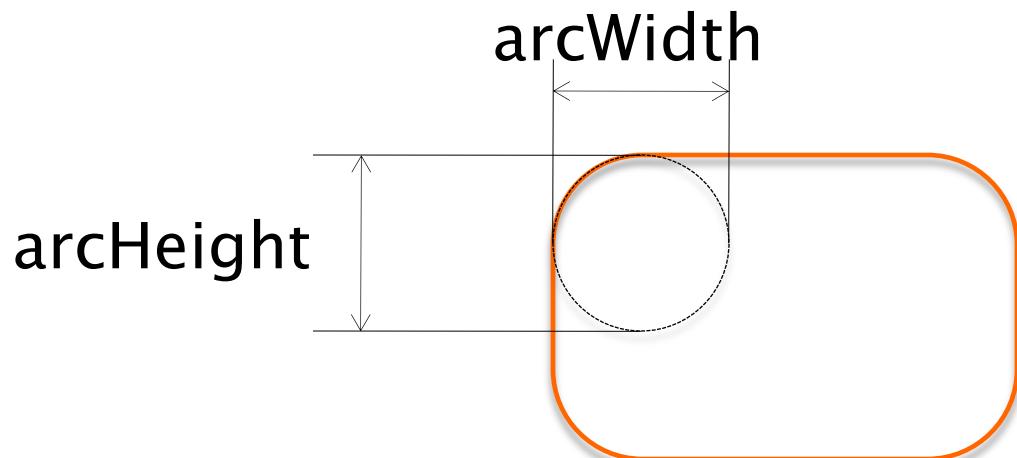


Graphics methods: lines

- **drawLine(x1,y1,x2,y2)**
 - Draw a line between two points
- **drawRect(x,y,width,height)**
 - Draw a rectangle (x,y) is upper left corner
 - Size is defined by width and height
- **fillRect()**
 - Same as above but rectangle is solid filled

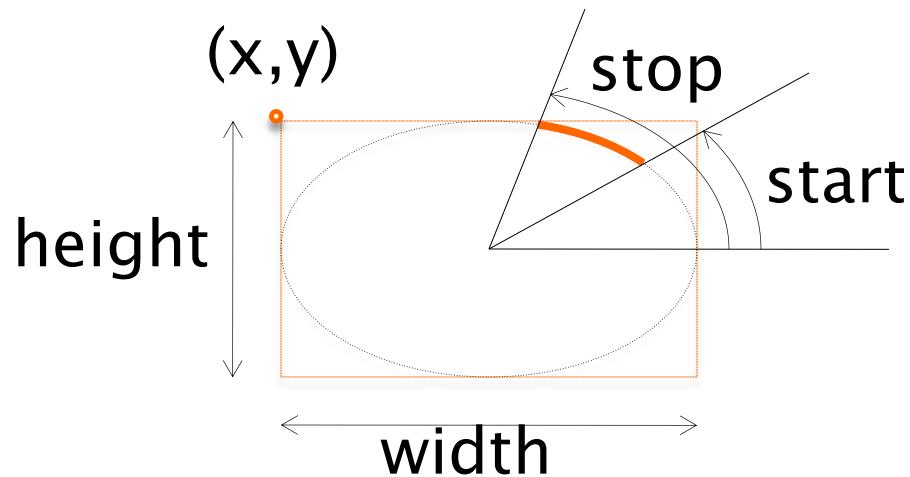
Graphics methods: rectangles

- **drawRoundRect(x, y, width, height, arcWidth, arcHeight)**
 - Draw a rectangle with rounded angles
- **fillRoundRect(x, y, width, height, arcWidth, arcHeight)**
 - Same as above but with solid filled shape



Graphics methods: ellipses

- **drawOval (x, y, width, height)**
 - Draw an ellipse inscribed in a rectangle located at (x,y) with the given size
- **drawArc (x, y, width, height, start, stop)**
 - Draw an arc of an ellipse starting at *start* degrees and stopping at *stop* degrees
- Also available:
 - **fillOval ()**
 - **fillArc ()**



Graphics methods: strings

- **drawString(str, x, y)**
 - Draw a string starting at point (x,y)
- **drawChars(chars, offset, length, x, y)**
 - Draw a char array starting at point (x,y)
 - Offset is the first char to draw
 - Length is the number of chars to draw

The string
(x,y)

Draw lines and squares

- To draw a line

```
g.drawLine(25, 25, 75, 75);
```

- To Draw a rectangle, specifying the coordinated point in the top left, width and length:

```
g.drawRect(20, 20, 60, 60);  
g.fillRect(120, 20, 60, 60);
```

- To Draw a rectangle, specifying the coordinated point in the top left, width and length:

```
g.drawRoundRect(20,20, 60,60, 10,10);  
g.fillRoundRect(120,20, 60,60, 20,20);
```

Draw polygons

- A polygon requires a set of points defined as two x and y arrays:

```
int x[ ] = {39,94,97,142,53,58, 26};  
int y[ ] = {33,74,36,70,108,80, 106};  
int points = x.length;  
g.drawPolygon(x,y,points);
```

- ..or as instances of the class polygon:

```
Polygon poly = new Polygon(x,y,points);  
g.fillPolygon(poly);
```

- The polygon is closed automatically
`drawPolyline()` allows to have open polygons.

Draw ellipses and arcs

- To draw circles or ellipses using the oval .

```
g.drawOval(20, 20, 60, 60);
```

```
g.fillOval(120, 20, 100, 60);
```

- Arcs are defined as pieces of ellipses with the method drawArc()
 - An ellipsis must be defined plus the starting and ending angles. Which are defined counterclockwise (90 vertical axis).

```
g.drawArc(20, 20, 60, 60, 90, 180);
```

```
g.fillArc(120, 20, 60, 60, 90, 180);
```

Draw strings

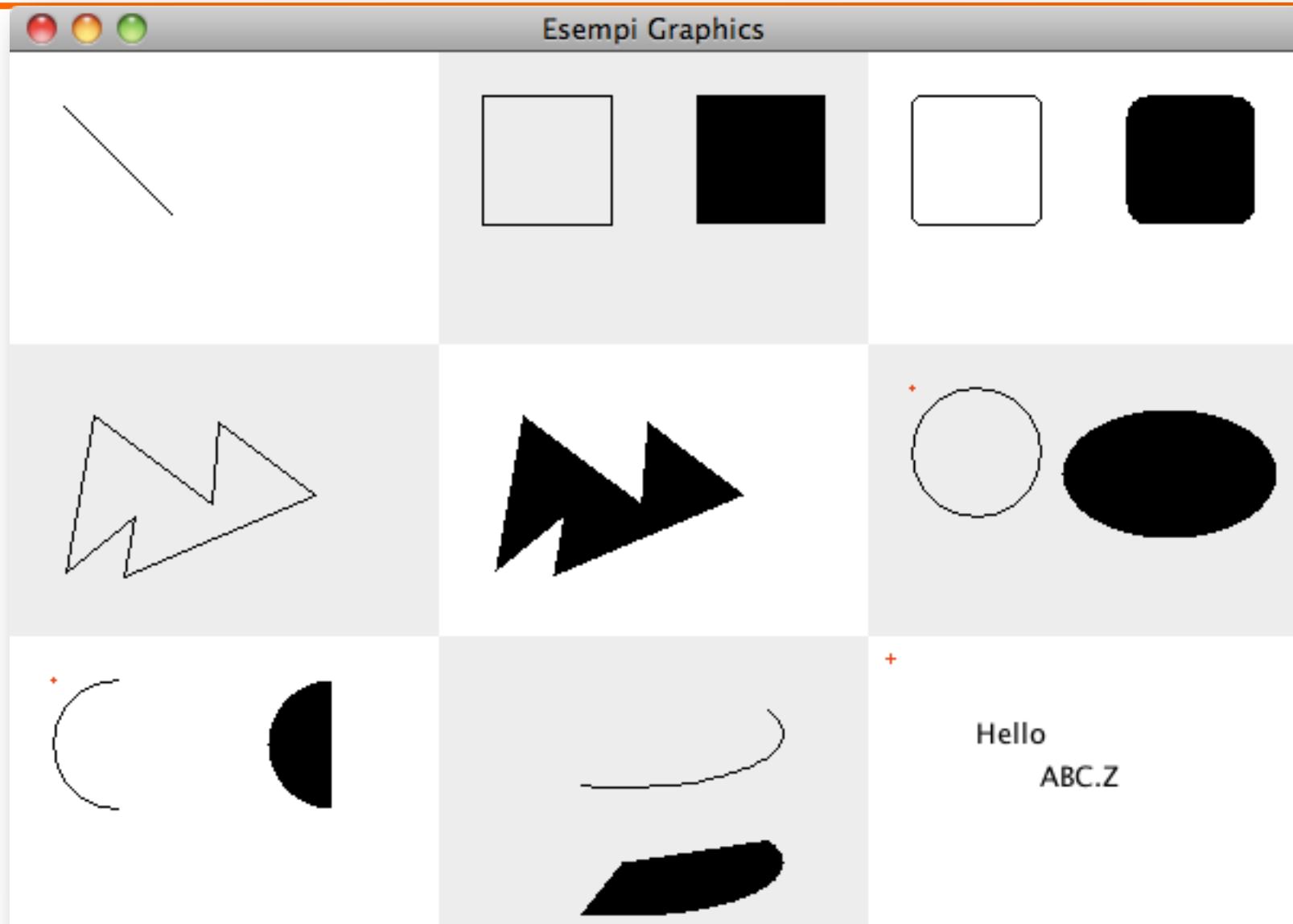
- To draw strings use:

```
g.drawString("Hello", 50, 50);
```

- ..or draw an array of chars:

```
char[] chars = new  
char[] {'A', 'B', 'C', '.', 'Z'};  
g.drawChars(chars, 0, chars.length, 80, 70);
```

Examples



Repaint

- Method **paint()** is invoked by OS when needed
 - E.g. window resize, de-iconify
- Method **repaint()** signals that window contents must be updated
 - Later OS will invoke paint()
- This method is essential to update the view when something is changed

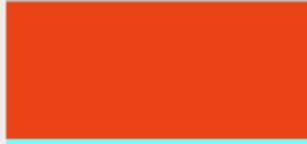
Color management

- The management of colors is performed through class Color.
 - Colors are encoded on 24 bit; each color consists of a combination of red, green and blue .
 - Each component is represented with a whole number between 0 and 255.
- There are class constants defined for the main colors.

Color management

- For windows (JFrame):
 - **setBackground(Color c)**
 - Sets the window internal background
 - **setForeground(Color c)**
 - Sets the components foreground color
- For Graphics:
 - **setColor(Color c)**
 - Sets the color for all the successive drawing operations

Color management

Colors					
Color	Code	Example	Color	Code	Example
Color.white	255, 255, 255		Color.black	0, 0, 0	
Color.lightGray	192, 192, 192		Color.darkGray	64, 64, 64	
Color.red	255, 0, 0		Color.green	0, 255, 0	
Color.cyan	0, 255, 255		Color.yellow	255, 255, 0	
Color.blue	0, 0, 255		Color.magenta	255, 0, 255	
Color.pink	255, 175, 175		Color.orange	255, 200, 0	

Font management

- Fonts are represented by class Font
- Constructor
 - `Font(String face, int attrs, int size)`
- Parameters
 - Face is the name of the font e.g. “TimesRoman”
 - Attrs represent attributes e.g. Font.BOLD
 - Size is expressed in points

Font management

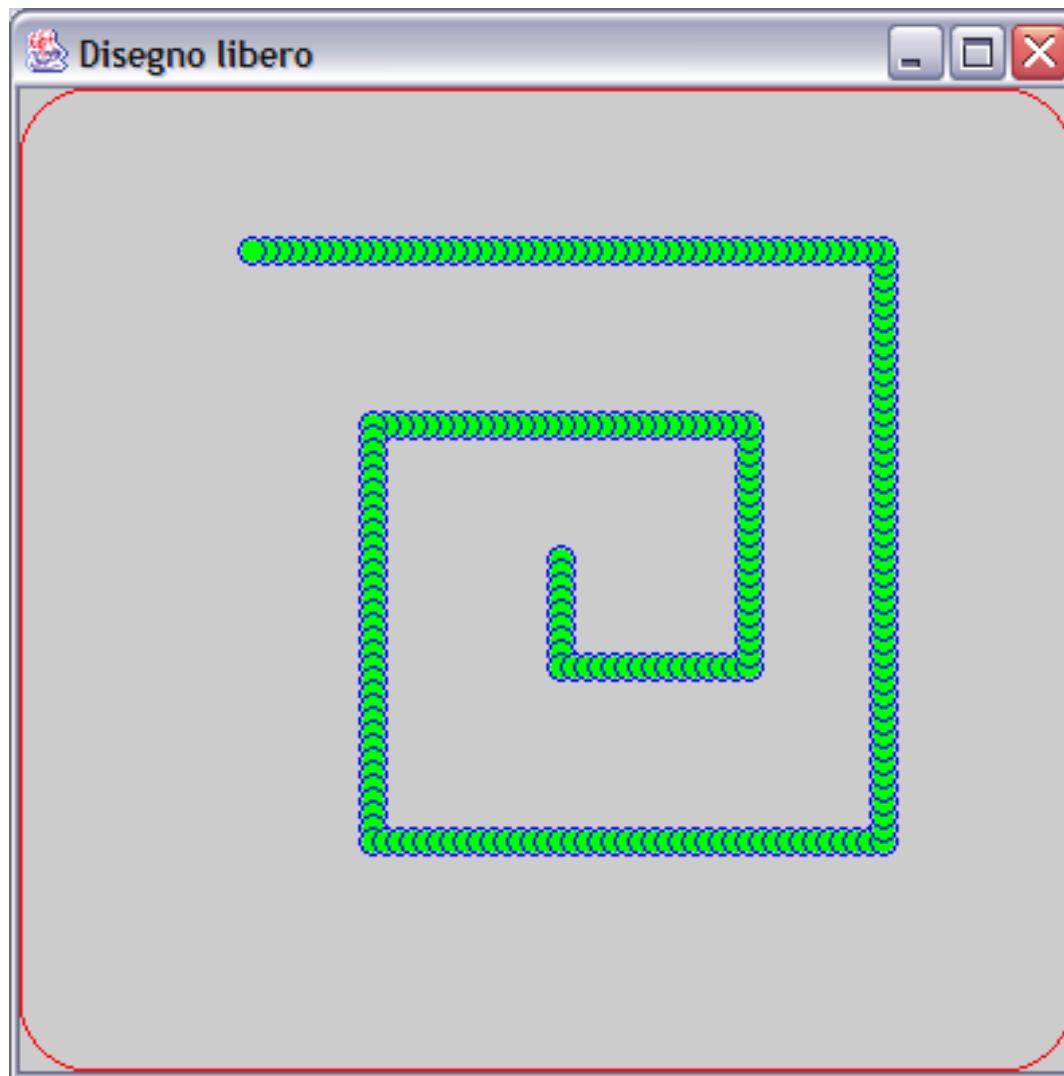
- To get information about a font:
 - `getFont()`: returns the current font
 - `getName()`: returns font name
 - `getSize()`: returns the font size
 - `getStyle()`: Return the style of font
 - `isPlain()`, `isBold()` , `isItalic()`: return the font modifications
- For more information more specific on the individual font use the class **FontMetrics**.

FontMetrics

- Main methods are:
 - `stringWidth()`: width in pixels of a given string
 - `charWidth()`: amplitude of a char
 - `getAscent()`
 - `getDescent()`
 - `getLeading()`
 - `getHeight()`

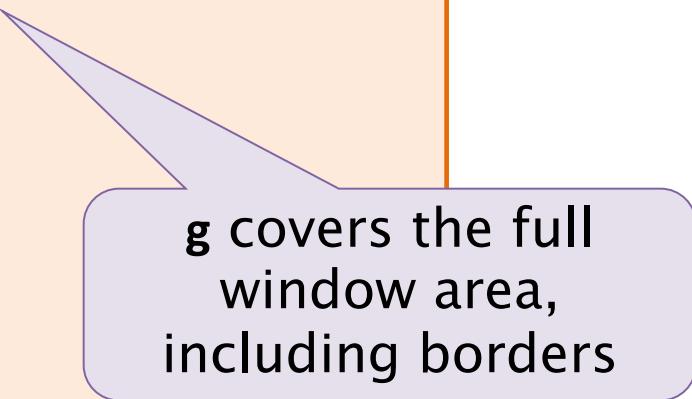


Example



Example

```
public class Drawing extends JFrame{  
    int x;  
    int y;  
    public void paint(Graphics g){  
        Rectangle b = getBounds();  
        g.setColor(Color.RED);  
        g.drawRoundRect(4,30,  
                        b.width-9,b.height-35,50,50);  
        g.setColor(Color.BLUE);  
        g.drawOval(x,y,10,10);  
        g.setColor(Color.GREEN);  
        g.fillOval(x+1,y+1,9,9);  
    }  
}
```

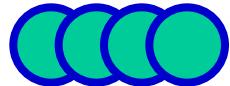


g covers the full
window area,
including borders

Events

```
public class Drawing implements KeyListener{  
    public void keyPressed(KeyEvent e) {  
        if(e.getKeyCode ()==KeyEvent.VK_DOWN) {  
            moveXY(0,5);  
        }  
        // ...  
    }  
    void moveXY(int deltaX, int deltaY){  
        x+=deltaX;  
        y+=deltaY;  
        this.repaint();  
    }  
}
```

Considerations

- The repaint operation does not erase the window
 - Therefore we have the trail effect 
- We need to explicitly erase the content of the window:

```
Rectangle bounds = getBounds() ;  
g.clearRect(0,0,bds.width,bounds.height) ;
```

Advices

- Define a method `paint` on an empty (e.g. without borders) component
- **DO NOT** override method `paint()` on a frame containing components
- Usually a `JPanel` is a good candidate to override method `paint()`

Summary

- GUI can be build using the MVC pattern:
 - Model: hosts the data
 - View: show the data
 - Controller: manages the interaction
- The view can be build using different libraries:
 - AWT
 - Swing
 - JavaFX

Summary

- In Swing the main elements are
 - JFrame that represent the view container
 - JComponent is the root class of all controls:
 - JButton
 - JLabel
 - JTextField
 - JPanel
 - ...

Summary

- The interaction takes place when an event is generated and managed by the appropriate listener
 - A listener must be registered for a component and a specific event category
 - When the event is generated the appropriate method of the listener is called back
 - The method can handle the event as required