Programmazione di reti Corso B

18 Maggio 2016
Lezione 12

Contenuti

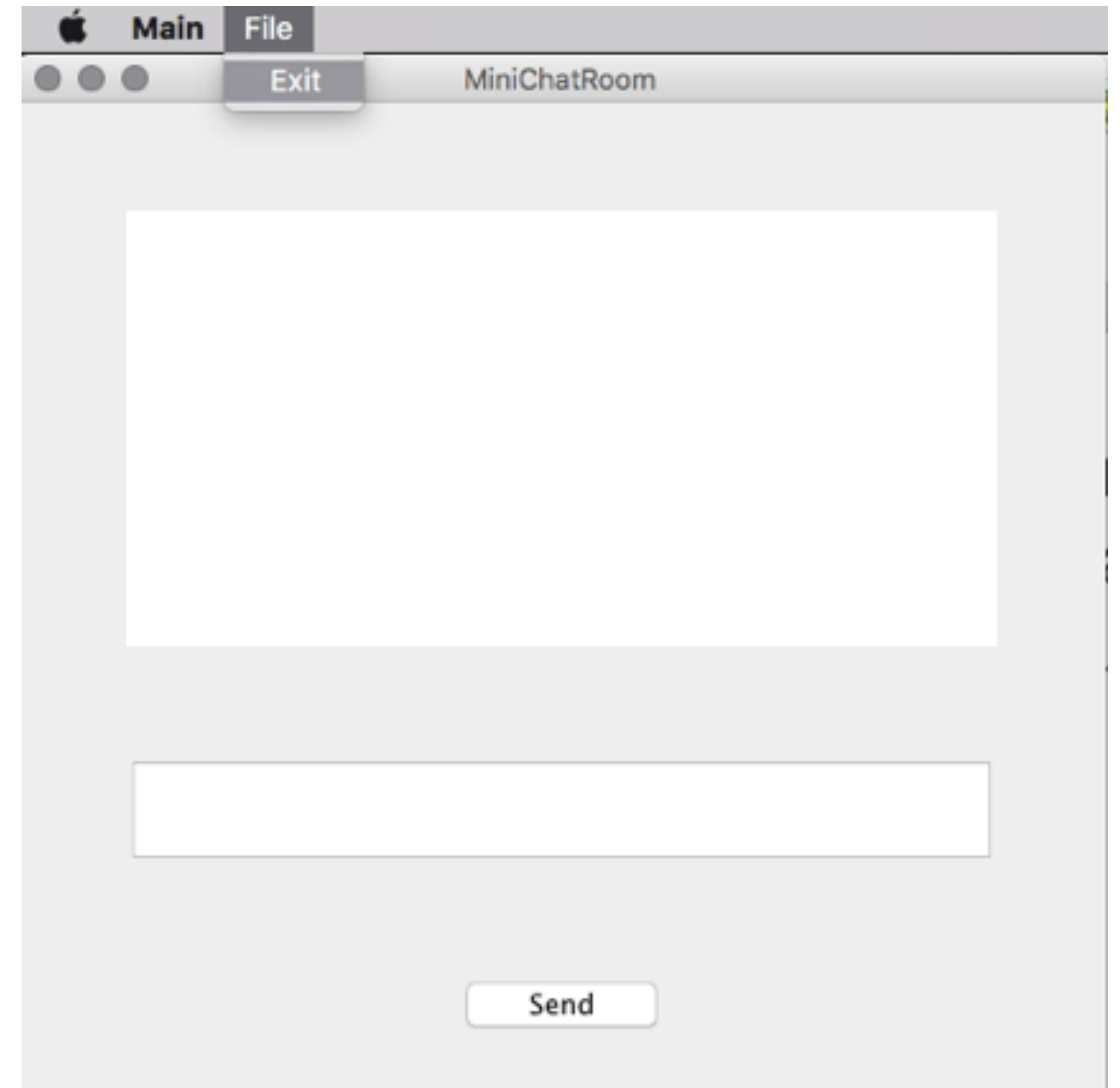
- `Java Swing`
  - contenitori
  - componenti
  - eventi e *multithreading*
  - dialoghi
  - menu
  - *layout*

Java Swing

- *Framework Java* per costruire applicazioni *desktop* con interfaccia grafica.
- Indipendente dalla piattaforma
- Implementata in *Java*
- 3 concetti principali:
    - componenti: elementi di interfaccia predefiniti che offrono la funzionalità all'utente in modo grafico.
    - comportamento: risposta a un set di eventi predefiniti per ogni componente
    - *layout*: modo in cui i componenti sono organizzati

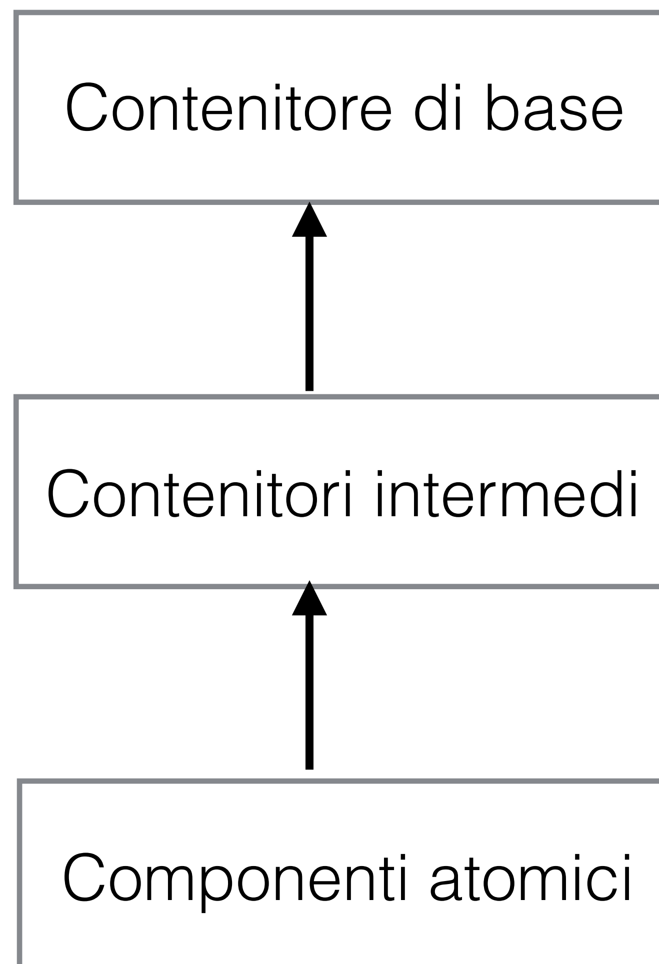
Struttura dell'interfaccia

- Vari componenti
  - Contenitori
    - Finestra
  - Componenti utente
    - pulsanti, aree di testo, menu



Gerarchia di componenti

- Relazione *parent-child*



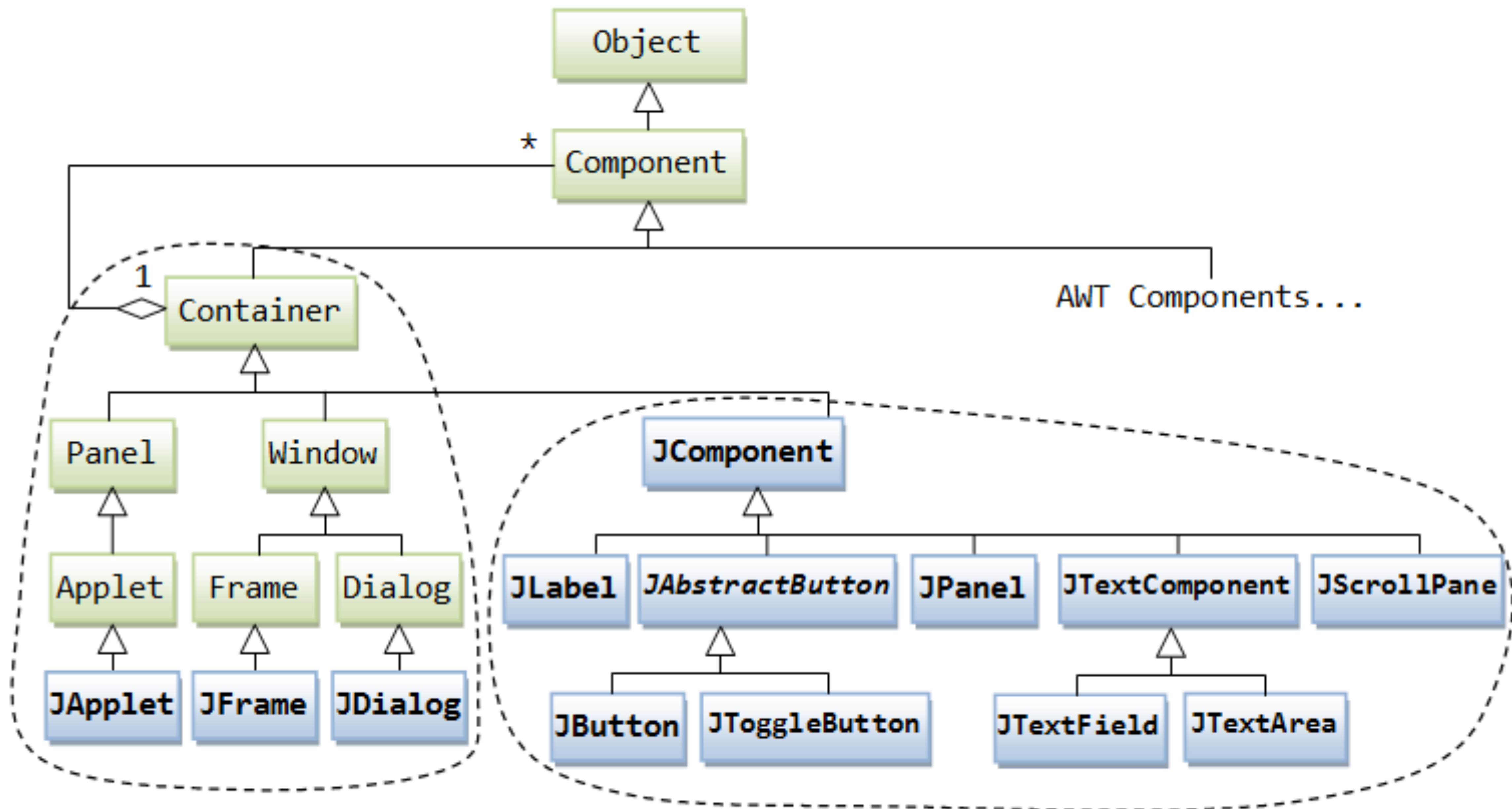
Gerarchia di componenti

- *Top-level container* (contenitore di base): `JFrame`, `JDialog`, `JApplet` - offrono l'ambiente dove le altre componenti si possono disegnare - *heavy-weight* - lavora con il sistema di finestre del SO

- *Intermediate container* (contenitore intermedio): `JRootPane`, `JScrollPane`, `JTabbedPane`, `JSplitPane`, `JPanel` - semplificano il posizionamento delle componenti

- Componenti atomici: `JButton`, `JTextField`, `JTable`, etc.

Gerarchia di classi

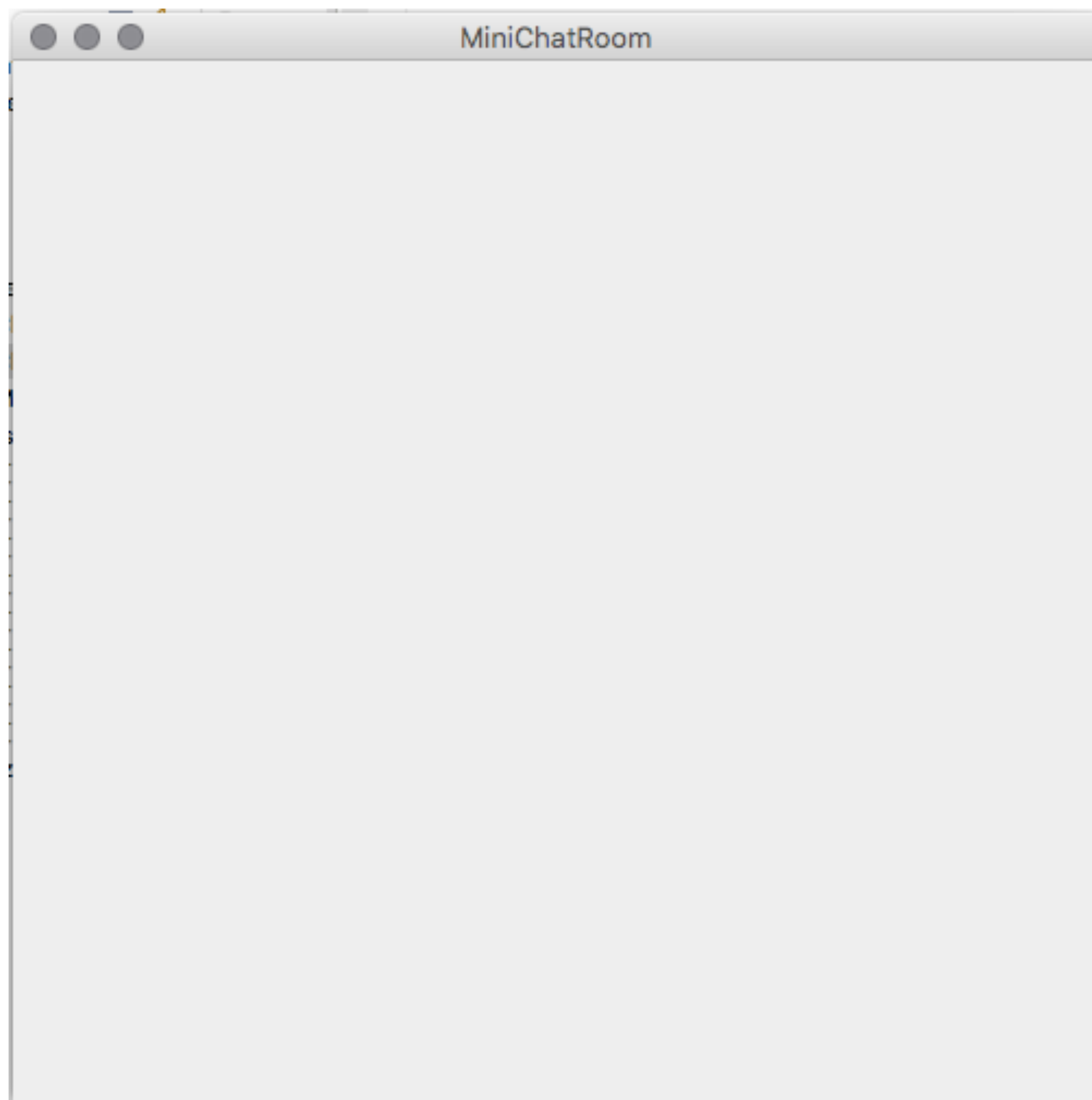


Finestre

- Contenitori di base: oggetti di tipo `JFrame`
- Contengono un contenitore intermedio di tipo `JRootPane`.
- Il *root pane* contiene il *content pane* - di tipo `JPanel`: contenitore per tutti gli elementi dell'interfaccia
- Per creare una finestra:
  - istanziare un'oggetto di tipo `JFrame`
  - dargli una dimensione in pixel con metodo `void setSize(int, int)`
  - farla visibile con il metodo `void setVisible(boolean)`
  - scegliere comportamento alla chiusura (fine programma, nascondi finestra, etc) `void setDefaultCloseOperation(int )`

```
public class Main {

   public static void main(String[] args) {
      ClientGui gui = new ClientGui();
      gui.setVisible(true);
   }

}
```

```
public class ClientGui extends JFrame {

   private static final long serialVersionUID = 1L;
   public ClientGui() {
      this.setSize(500, 500);
      this.setTitle(“MiniChatRoom");
      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
   }
}
```


MiniChatRoom

Componenti

- Implementano la classe `Component`
- Swing contiene componenti predefiniti.
- La finestra e anche lei un `Component`
- Vari componenti dell'interfaccia possono essere aggiunti alla finestra usando metodo

```
void add(Component c)
```

- Ogni componente può contenere altri componenti

Component

- Vari metodi per aggiungere altri componenti e per formattare i componenti

`void add (Component x)`

`x` diventa child per questa componente

`void setLocation(int x, int y)`

Posizione dell'angolo in alto a sinistra in pixel, in relazione alla componente parent.

`void setSize(int width, int height)`

Dimensione in pixel.

`void setVisible(boolean b)`

Stato visibile/invisibile della componente

Component

- Vari metodi per formattare i componenti

```
void   setFont(Font f)

void   setForeground(Color c)

void   setBackground(Color c)

void repaint()
```

Aggiorna la visualizzazione.

```
void   setEnabled(boolean b)
```

Abilita la componente di ricevere input dall'utente tramite mouse, keyboard.

JButton

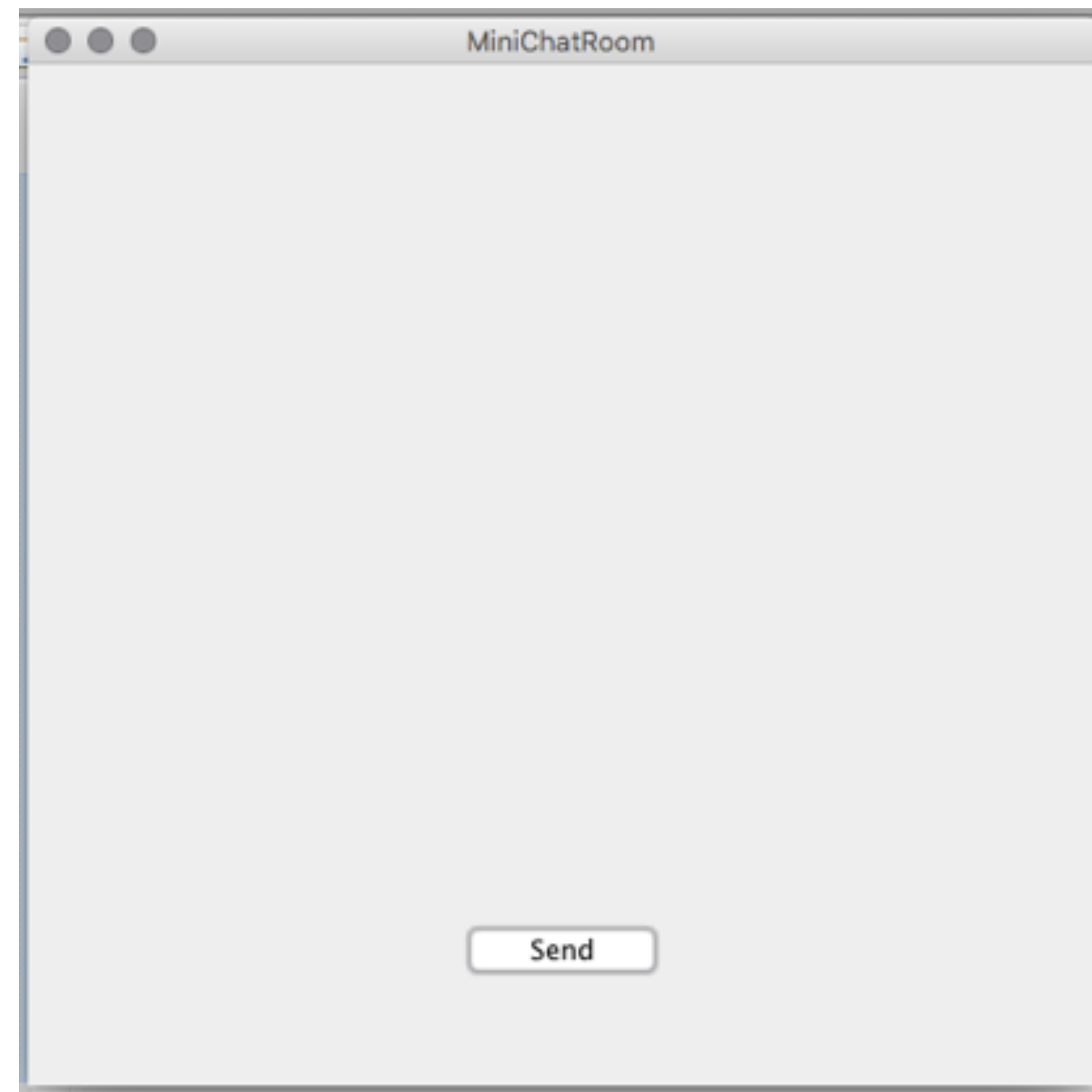
- Clase usata per creare pulsanti.
- Costruttori:

```
JButton(Icon icon)
JButton(String text)
JButton(String text, Icon icon)
```

`icon` può essere definito usando un'immagine:

```
Icon icon =new ImageIcon(filename);
```

```java
public class ClientGui extends JFrame {

    private static final long serialVersionUID = 1L;
    public ClientGui() {
        this.setSize(500, 500);
        this.setTitle("MiniChatRoom");
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setLayout(null);

        JButton sendButton= new JButton("Send");
        this.add(sendButton);
        sendButton.setLocation(200, 400);
        sendButton.setSize(100, 30);


    }
}
```



Testo

- Componenti `JTextField` e `JTextArea`, estendono `JTextComponent`
- Vari metodi per accedere a contenuto e eventi

```
String  getText()
```

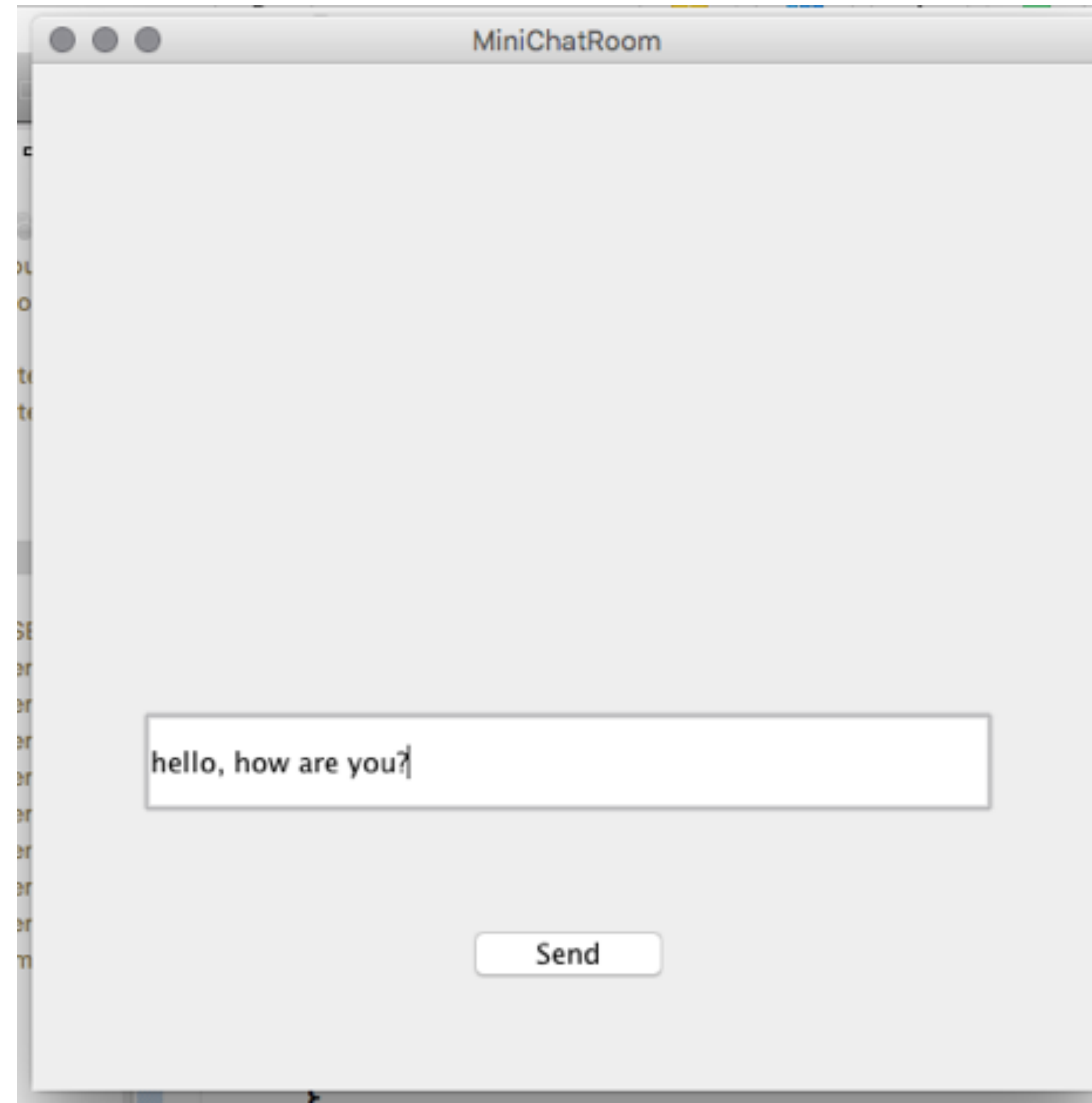
```
String  setText()
```

Restituisce o cambia il testo della componente

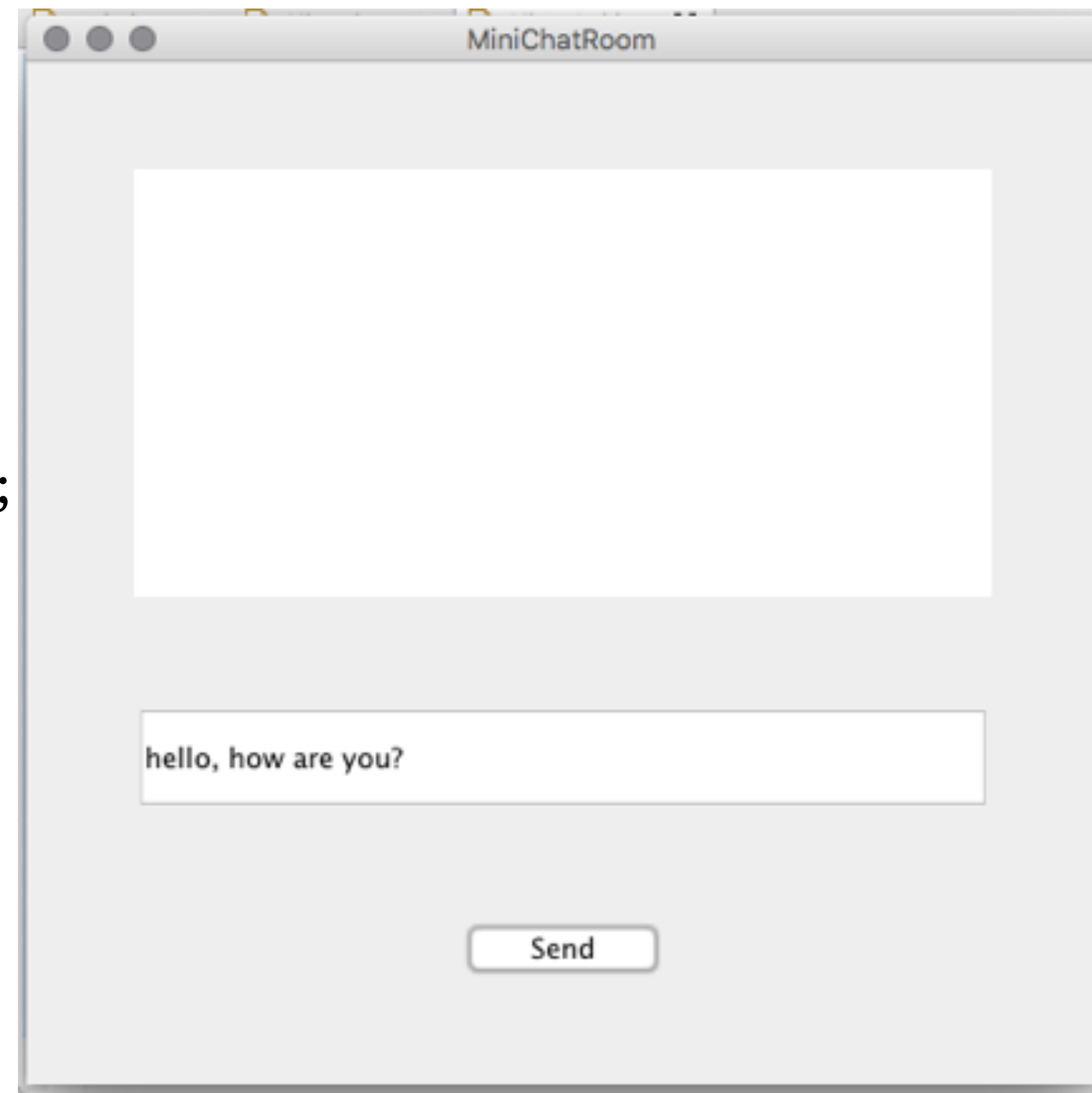
```
void setEditable(boolean)
```

Abilita l’utente di cambiare il testo

```java
public class ClientGui extends JFrame {

   private static final long serialVersionUID = 1L;
   public ClientGui() {
      this.setSize(500, 500);
      this.setTitle("MiniChatRoom");
      this.setDefaultCloseOperation(EXIT_ON_CLOSE);
      this.setLayout(null);

      JButton sendButton= new JButton("Send");
      this.add(sendButton);
      sendButton.setLocation(200, 400);
      sendButton.setSize(100, 30);

      JTextField inputArea= new JTextField();
      this.add(inputArea);
      inputArea.setLocation(50, 300);
      inputArea.setSize(400, 50);

   }
}
```



```java
public class ClientGui extends JFrame {

    private static final long serialVersionUID = 1L;
    public ClientGui() {
        this.setSize(500, 500);
        this.setTitle("MiniChatRoom");
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setLayout(null);

        JButton sendButton= new JButton("Send");
        this.add(sendButton);
        sendButton.setLocation(200, 400);
        sendButton.setSize(100, 30);

        JTextField inputArea= new JTextField();
        this.add(inputArea);
        inputArea.setLocation(50, 300);
        inputArea.setSize(400, 50);

        JTextArea receivedMessagesArea= new JTextArea();
        this.add(receivedMessagesArea);
        receivedMessagesArea.setLocation(50, 50);
        receivedMessagesArea.setSize(400, 200);
        receivedMessagesArea.setEditable(false);
    }
}
```



Eventi

- Ogni volta che lo stato di una componente cambia, viene lanciato un evento.
- Gli eventi vengono generati in modo asincrono quando l'utente interagisce con l'interfaccia (muove il *mouse*, *click* su una componente, *scroll*, etc)
- Ogni evento ha una *source* (componente che ha cambiato stato), e un *listener* - o *event handler* - (definisce azioni da eseguire quando l'evento succede)
- Il programmatore deve definire il *listener* e associarlo ad una componente.
- Ogni componente definisce i suoi eventi e accetta dei *listener* specializzati - tutti implementano interfaccia `EventListener`

Interfaccia `ActionListener`

- Interfaccia di base per *listener* accettati su pulsanti
- Definisce metodo da eseguire quando il pulsante è premuto

```
void actionPerformed(ActionEvent e)
```

- `ActionEvent` e una classe che contiene informazioni sull'evento - incluso la componente su quale l'evento è stato generato (metodo `Object getSource()`)
- Per associare un `ActionListener` ad un `JButton`, si usa il metodo della classe `JButton`

```
void addActionListener(ActionListener)
```

`MiniChatRoom`

- Il pulsante *Send* deve inviare il messaggio al *server*
- Nella classe `ClientGui` abbiamo bisogno di un riferimento a una classe che gestisce le connessioni e messaggi - classe `Client`
- Dobbiamo implementare un `ActionListener` per il pulsante - anche questo oggetto contiene riferimento al `Client`
- Dopo aver creato pulsante e cliente, dobbiamo associare il *listener* al pulsante

- Implementazione della classe `ClientGui`:

```java
public class ClientGui extends JFrame {

    private static final long serialVersionUID = 1L;

    private  JTextArea receivedMessagesArea;
    public  JTextArea getReceivedMessagesArea(){
        return receivedMessagesArea;
    }

    private  JTextField inputArea;
    public  JTextField getInputArea(){
        return inputArea;
    }

    private Client client;

    public ClientGui() {
        this.setSize(500, 500);
        this.setTitle("MiniChatRoom");
        this.setDefaultCloseOperation(EXIT_ON_CLOSE);
        this.setLayout(null);

        JButton sendButton= new JButton("Send");
        this.add(sendButton);
        sendButton.setLocation(200, 400);
        sendButton.setSize(100, 30);
```

```java
        inputArea= new JTextField();
        this.add(inputArea);
        inputArea.setLocation(50, 300);
        inputArea.setSize(400, 50);

        receivedMessagesArea= new JTextArea();
        this.add(receivedMessagesArea);
        receivedMessagesArea.setLocation(50, 50);
        receivedMessagesArea.setSize(400, 200);
        receivedMessagesArea.setEditable(false);

        try {
            this.client= new Client(this);
            sendButton.addActionListener(new SendActionListener(
                    client));
        } catch (IOException e) {
            receivedMessagesArea.setText("Some error occured connecting to server.");
            e.printStackTrace();
        }
    }
}
```

- Implementazione del listener

```java
public class SendActionListener implements ActionListener{

    Client client;

    public  SendActionListener(Client c) {
        this.client=c;
    }
    @Override
    public void actionPerformed(ActionEvent e) {
        JButton button=(JButton) e.getSource();
        ClientGui gui= (ClientGui) button.getTopLevelAncestor();
        JTextField message=gui.getInputArea();
        JTextArea history=gui.getReceivedMessagesArea();
        String text=message.getText();
        try {
            this.client.send(text);
            message.setText("");
            history.setText(history.getText()+"\nme: "+text);
        } catch (IOException e1) {
            history.setText(history.getText()+"\nme: "+text +" - maybe not sent");
            e1.printStackTrace();
        }
    }

}
```

- Implementazione della classe `Client`:
    - Apre connessione TCP al *server*
    - Apre *socket multicast* - avvia un *thread* separato per ricevere i messaggi
    - Metodo `void send(String)` invia messaggi al *server* - viene richiamato dal *listener*

```java
public class Client {
    private static final int SERVER_PORT=2001;
    private static final String SERVER_HOST="localhost";
    private static final int MULTICAST_PORT=2000;
    private static final String MULTICAST_GROUP="239.255.2.2";

    private Socket tcpSocket=null;
    private MulticastSocket udpSocket=null;
    private ClientGui gui;

    public Client(ClientGui clientGui) throws UnknownHostException, IOException{
        this.gui= clientGui;
        this.tcpSocket=new Socket(SERVER_HOST, SERVER_PORT);
        this.udpSocket= new MulticastSocket(MULTICAST_PORT);
        this.udpSocket.joinGroup(InetAddress.getByName(MULTICAST_GROUP));

        ExecutorService es= Executors.newSingleThreadExecutor();
        ClientReceiver receiver= new ClientReceiver(this.udpSocket,this.gui);
        es.submit(receiver);
    }
```

```java
    public void send(String message) throws IOException{
        BufferedWriter writer = new BufferedWriter(new OutputStreamWriter(
                tcpSocket.getOutputStream(),"UTF-16"));
        writer.write(message+"#");
        writer.flush();
    }

    public void close() {
        try{
            if(this.tcpSocket!=null)
                this.tcpSocket.close();
            if(this.udpSocket!=null)
                this.udpSocket.close();
        } catch(IOException e){
            System.out.println(e.getMessage());
        }
    }
}
```

Gestione eventi

- Tutti gli eventi vengono gestiti in un *thread* separato in modo asincrono- *event-dispatching thread* - esegue il codice degli *event-listener*

- Di solito la gestione di un evento include modifiche dello stato dei componenti

- I **componenti *Swing* non sono *thread-safe*** - tutte le modifiche dell'interfaccia devono essere fate da un solo *thread*- l'*event-dispatching thread*

- *Swing* offre la possibilità di usare l'*event-dispatching thread* per avviare qualsiasi codice che modifica lo stato dei componenti (non solo per i *listener*)

Multithreading

- Per la maggior parte delle applicazioni, tutti i cambi di stato dell'interfaccia si possono fare usando gli *event-listener*, quindi il programmatore non gestisce l'interfaccia da altri *thread*.
- A volte però serve poter cambiare lo stato anche su eventi esterni: e.g. in rete, i messaggi in arrivo sono asincroni, e devono essere visualizzati
- Si può usare un altro *thread* per aspettare i messaggi, però gli update dell'interfaccia si devono eseguire nel *event-dispatching thread*

Multithreading

- La classe `SwingUtils` ci aiuta:
- metodo `void invokeLater(Runnable)`
- esegue il metodo `run()` del `Runnable` nel *event-dispatching thread* di *Swing*. Restituisce subito.
- metodo `void invokeAndWait(Runnable )`
- Come `invokeLater` però aspetta che il metodo `run()` sia eseguito prima di continuare. E' consigliato di usarlo il meno possibile (possibilità di *deadlock*).

MiniChatRoom

- *Thread* separato che aspetta messaggi *multicast* - creammo una classe `ClientReceiver` che implementa `Runnable`
- Quando il messaggio arriva, il *thread* deve includerlo nella area di testo usando `receivedMessagesArea.setText(message)`
- Questo commando deve essere eseguito nel *event-dispatching thread*, quindi deve essere incluso nel metodo `run()` di un altro `Runnable` (classe `MessageShower`)

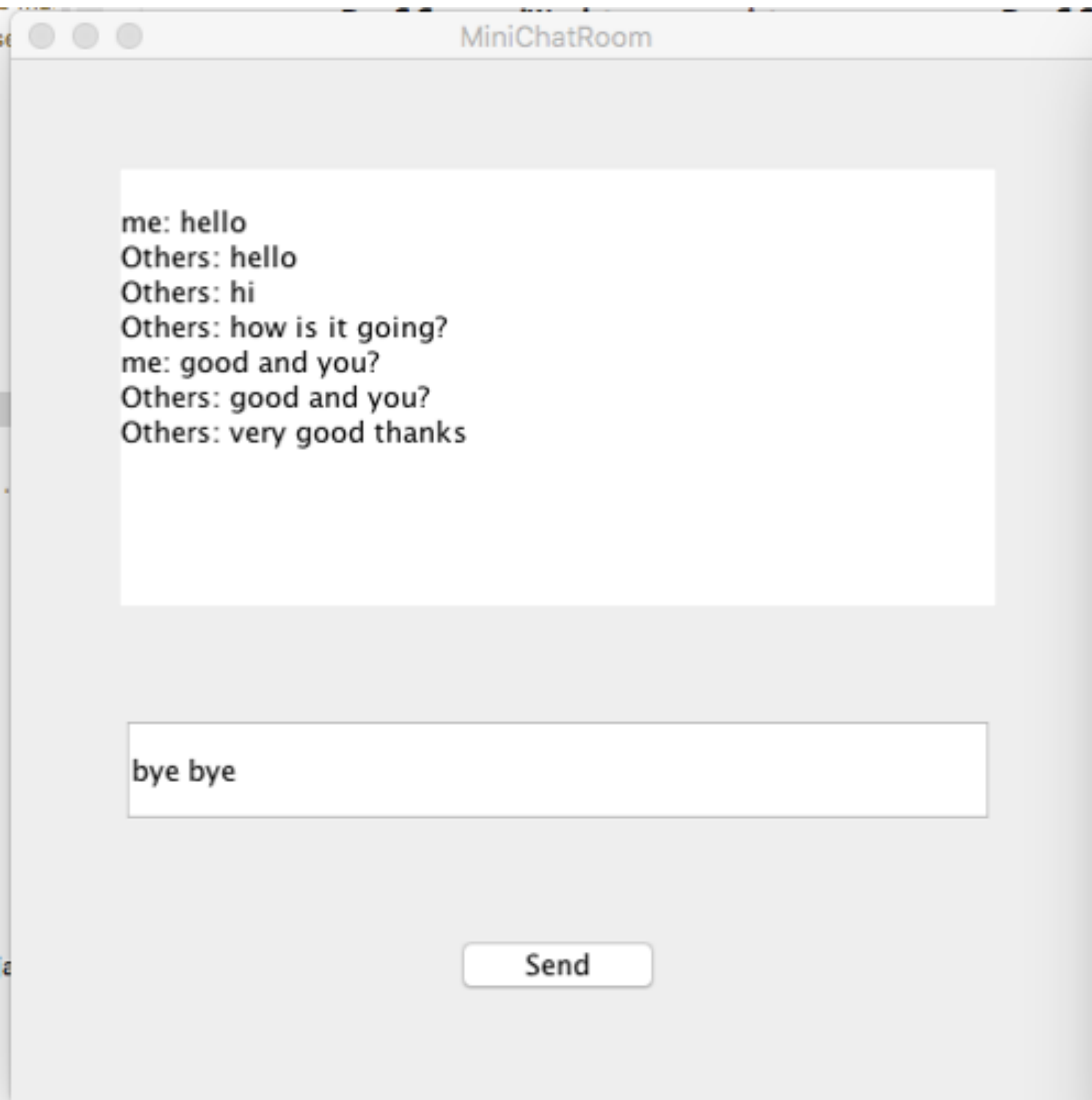
```java
public class ClientReceiver implements Runnable{

    MulticastSocket socket;
    ClientGui gui;

    public ClientReceiver(MulticastSocket udpSocket, ClientGui gui) {
        this.socket=udpSocket;
        this.gui=gui;
    }

    @Override
    public void run() {
        while(true){
            byte[] buffer= new byte[512];
            DatagramPacket packet = new DatagramPacket(buffer, 512);
            try{
                this.socket.receive(packet);
                String message= new String(buffer,0,packet.getLength(),"UTF-16");
                MessageShower shower = new MessageShower(this.gui,message);
                SwingUtilities.invokeLater(shower);
            } catch(IOException e){
                MessageShower shower = new MessageShower(this.gui,
                        "Error receiving message");
                SwingUtilities.invokeLater(shower);
            }
        }
    }
}
```

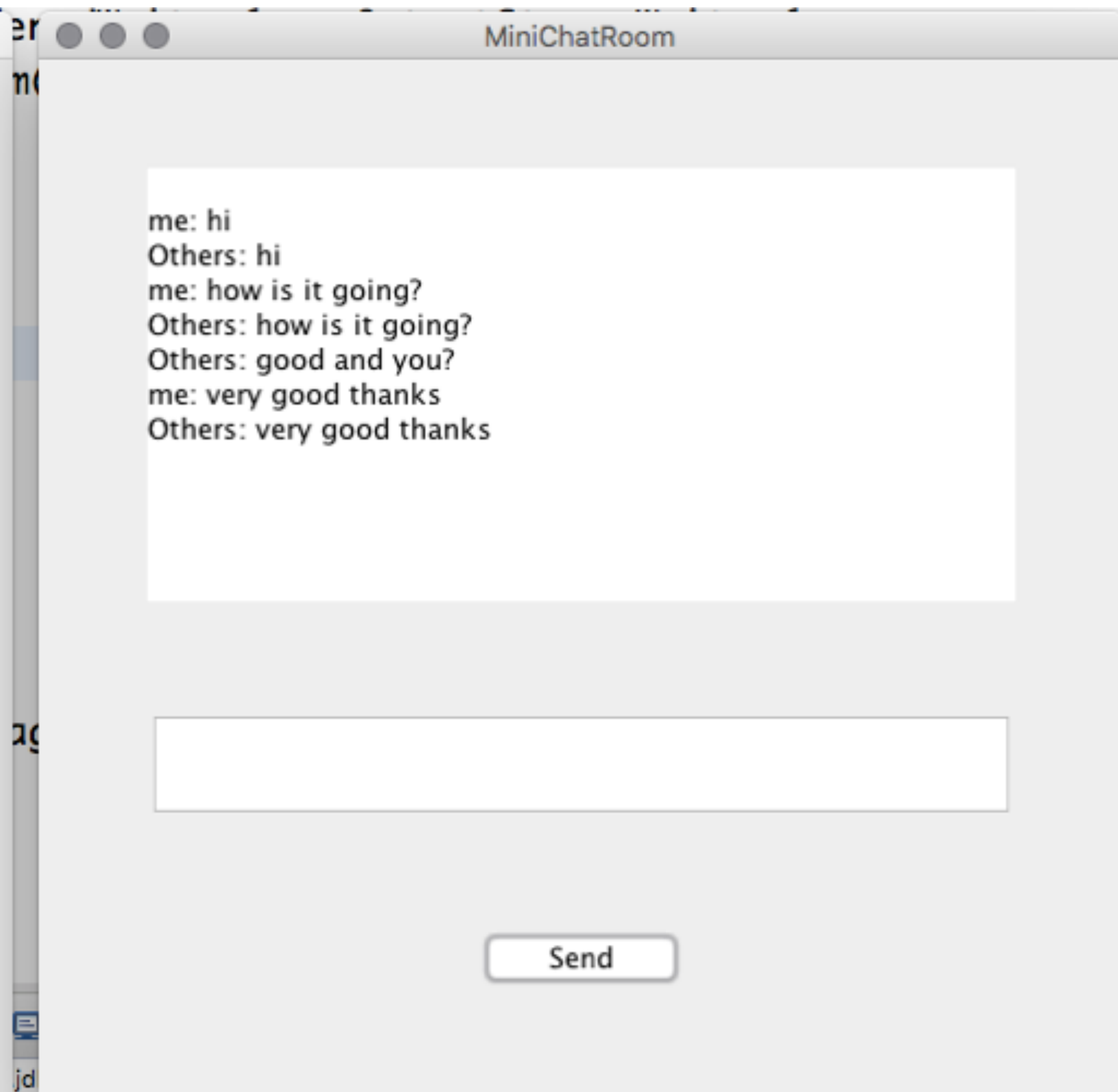
```java
public class MessageShower implements Runnable {

    ClientGui gui;
    String message;

    public MessageShower(ClientGui gui, String message) {
        this.gui=gui;
        this.message=message;
    }

    @Override
    public void run() {
        this.gui.getReceivedMessagesArea().setText(
                this.gui.getReceivedMessagesArea().getText()+
                "\nOthers: "+this.message);
    }

}
```

Multithreading

- Anche la parte di inizializzazione dell'interfaccia dovrebbe essere eseguita nel *thread* di *event-dispatching*
- La classe `Main` deve essere aggiustata:

```java
public class Main implements Runnable{

   public static void main(String[] args) {
      SwingUtilities.invokeLater(new Main());
   }

   @Override
   public void run() {
      ClientGui gui = new ClientGui();
      gui.setVisible(true);

   }

}
```






```
New client.
waiting for clients...
Received and forwarding: hi
Received and forwarding: how is it going?
Received and forwarding: good and you?
Received and forwarding: very good thanks
```

Event-listener

- `ActionListener` usato per pulsanti (si può usare anche per altre componenti se definito dalla componente)
- Ogni componente può accettare diversi *listener*
- Tutti i componenti accettano:
  - `KeyListener`

```
void   keyPressed(KeyEvent e)

void   keyReleased(KeyEvent e)

void   keyTyped(KeyEvent e)
```

Event-listener

- Tutti i componenti accettano:
  - `MouseListener`

```
void  mouseClicked(MouseEvent e)
void  mouseEntered(MouseEvent e)
void  mouseExited(MouseEvent e)
void  mousePressed(MouseEvent e)
void  mouseReleased(MouseEvent e)
```

  - `FocusListener`

```
void focusGained(FocusEvent e)
void focusLost(FocusEvent e)
```

Event-listener

- Tutti i componenti accettano:
  - `MouseMotionListener`

    ```
    void  mouseDragged(MouseEvent e)
    void  mouseMoved(MouseEvent e)
    ```
  - `MouseWheelListener`

    ```
    void  mouseWheelMoved(MouseWheelEvent e)
    ```

Event-listener

- Le finestre (`JFrame`, `JDialog`) accettano:
  - `WindowListener`

```
void  windowActivated(WindowEvent e)
void  windowClosed(WindowEvent e)
void  windowClosing(WindowEvent e)
void  windowDeactivated(WindowEvent e)
void  windowDeiconified(WindowEvent e)
void  windowIconified(WindowEvent e)
void  windowOpened(WindowEvent e)
```

Event-listener

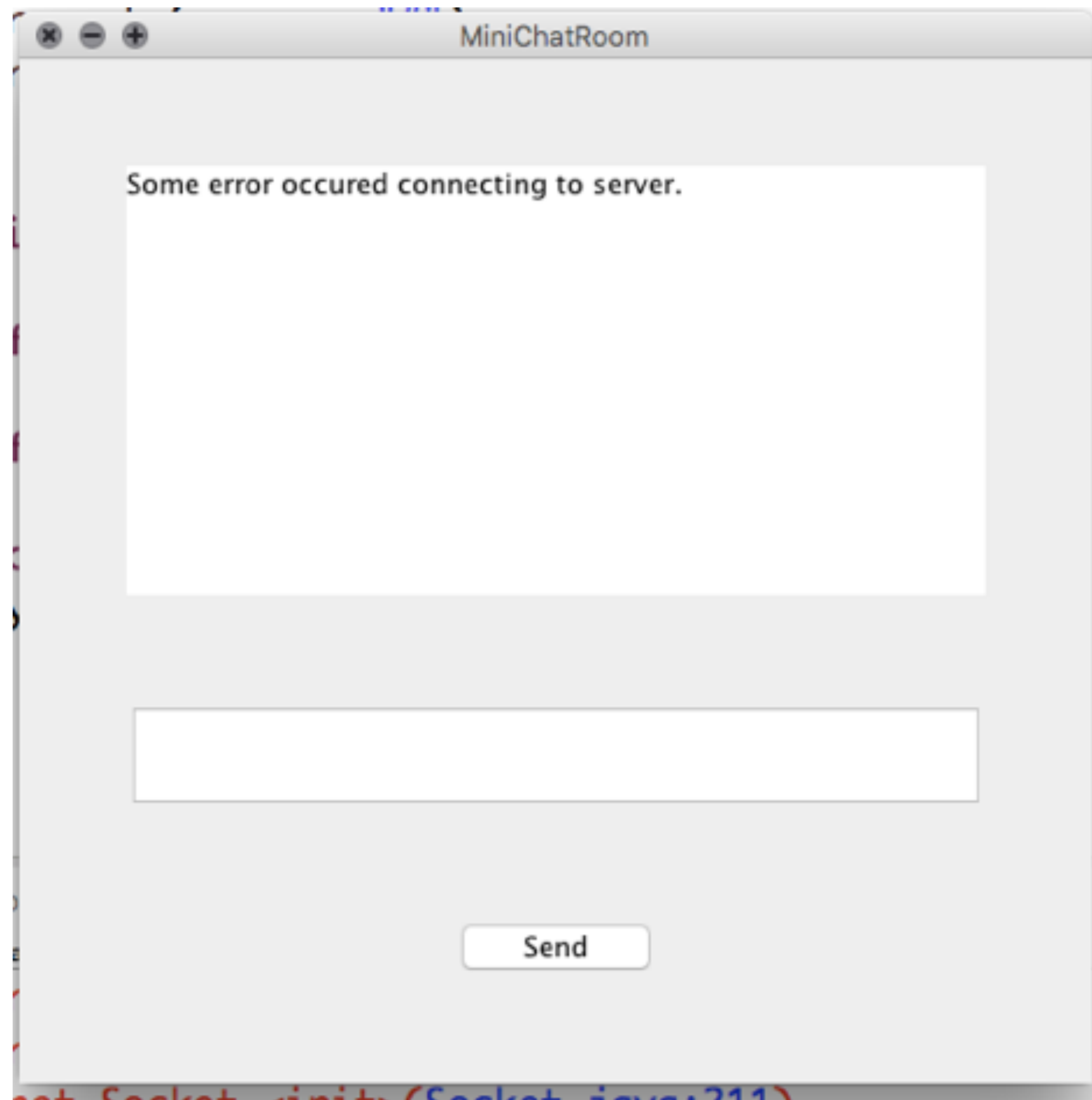
- Le finestre (`JFrame, JDialog`) accettano:
  - `WindowFocusListener`

```
void windowGainedFocus(WindowEvent e)

void windowLostFocus(WindowEvent e)
```

Finestre di dialogo

- Componenti di base (top-level) - implementate nella classe `JDialog`
- Usate per notificare l'utente di un evento, per ricevere input dall'utente, etc
- `MiniChatRoom`- potremmo usarlo in caso di errore, per notificare l'utente e uscire dall'applicazione

Adesso facciamo solo:




```
catch (IOException e) {
        receivedMessagesArea.setText("Some error occured connecting to server.");
        e.printStackTrace();
    }
```

Finestre di dialogo

- Contiene un `JRootPane` e un *content pane* dove si possono aggiungere altri componenti
- Costruttori:
    - `JDialog()`
    - `JDialog(Dialog owner, String title, boolean modal)`
    - `JDialog(Frame owner, String title, boolean modal)`
    - `JDialog(Window owner, String title, Dialog.ModalityType modalityType)`
    - etc
    - `modal=true` : la finestra di dialogo blocca le altre finestre dell’applicazione

- `MiniChatRoom` - finestra di dialogo per errore

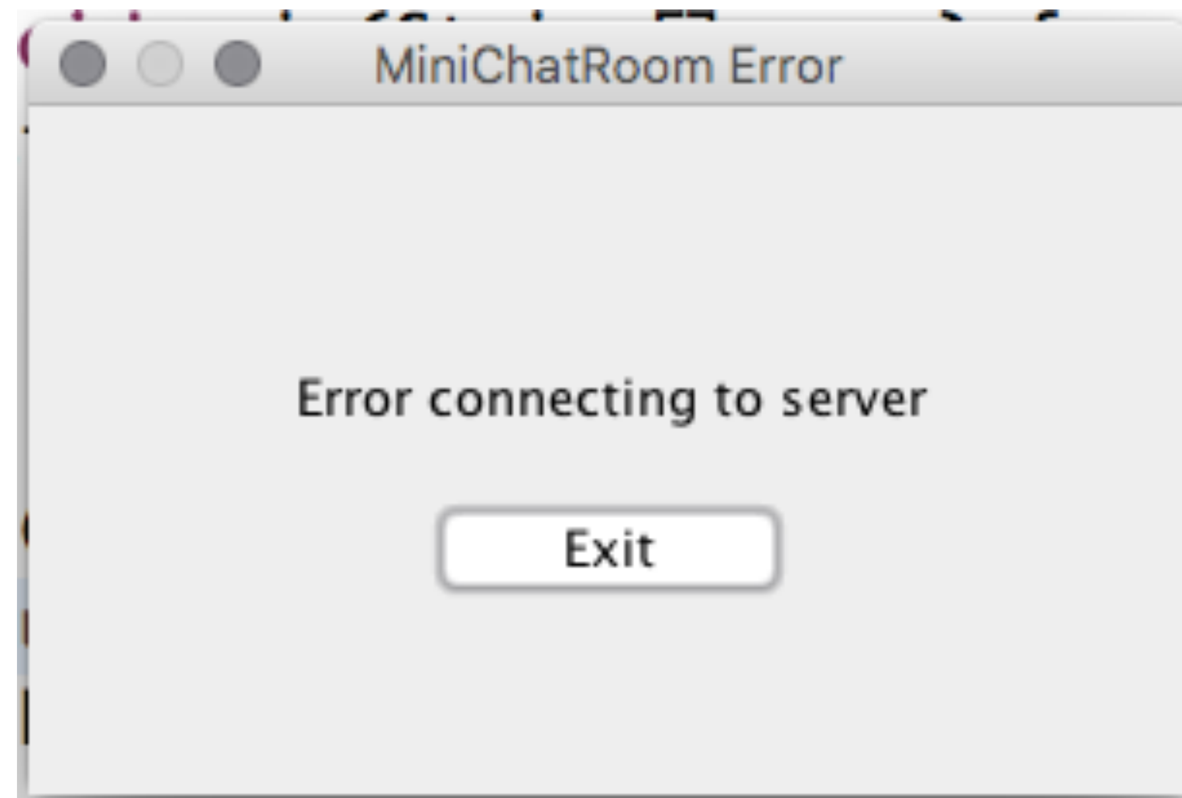
```java
public class FatalErrorDialog extends JDialog{
   private static final long serialVersionUID = 1L;
   String message;
   Client client;

   public FatalErrorDialog(ClientGui parent, Client client, String message ) {
      super(parent,"MiniChatRoom Error",true);
      this.message=message;
      this.client=client;
      this.setLayout(null);
      this.setSize(300,200);

      JLabel messageLabel= new JLabel(this.message);
      this.add(messageLabel);
      messageLabel.setLocation(30, 50);
      messageLabel.setSize(240,50);
      messageLabel.setHorizontalAlignment(SwingConstants.CENTER);
      messageLabel.setVerticalAlignment(SwingConstants.CENTER);

      JButton exitButton= new JButton("Exit");
      this.add(exitButton);
      exitButton.setSize(100,30);
      exitButton.setLocation(100,100);

      exitButton.addActionListener(new ExitButtonListener(this.client) );
      this.addWindowListener(new ExitWindowListener(this.client));
   }}
```

```java
public class ExitButtonListener implements ActionListener{

      private Client client;

      public ExitButtonListener(Client client) {
         this.client=client;
      }

      @Override
      public void actionPerformed(ActionEvent e) {
         if (client!=null)
            client.close();
         System.exit(0);
      }

   }
```

```java
public class ExitWindowListener implements WindowListener{

    private Client client;

    public ExitWindowListener(Client client) {
        this.client=client;
    }
    @Override
    public void windowOpened(WindowEvent e) {}
    @Override
    public void windowClosing(WindowEvent e) {
        if (client!=null)
            this.client.close();
        System.exit(0);
    }
    @Override
    public void windowClosed(WindowEvent e) {}
    @Override
    public void windowIconified(WindowEvent e) {}
    @Override
    public void windowDeiconified(WindowEvent e) {}
    @Override
    public void windowActivated(WindowEvent e) {}
    @Override
    public void windowDeactivated(WindowEvent e) {}
}
```

- Costruttore di `ClientGui` diventa:

```java
public ClientGui() {
    this.setSize(500, 500);
    this.setTitle("MiniChatRoom");
    this.setLayout(null);

    JButton sendButton= new JButton("Send");
    this.add(sendButton);
    sendButton.setLocation(200, 400);
    sendButton.setSize(100, 30);

    inputArea= new JTextField();
    this.add(inputArea);
    inputArea.setLocation(50, 300);
    inputArea.setSize(400, 50);

    receivedMessagesArea= new JTextArea();
    this.add(receivedMessagesArea);
    receivedMessagesArea.setLocation(50, 50);
    receivedMessagesArea.setSize(400, 200);
    receivedMessagesArea.setEditable(false);

    try {
        this.client= new Client(this);
        sendButton.addActionListener(new SendActionListener(client));
        this.addWindowListener(new ExitWindowListener(client));
    } catch (IOException e) {
        FatalErrorDialog d= new FatalErrorDialog(this,
                this.client, "Error connecting to server");
        d.setVisible(true);
    }}
```


MiniChatRoom Error
Error connecting to server
Exit

Menu

- *Swing* include delle componenti per creare un menu nell'applicazione

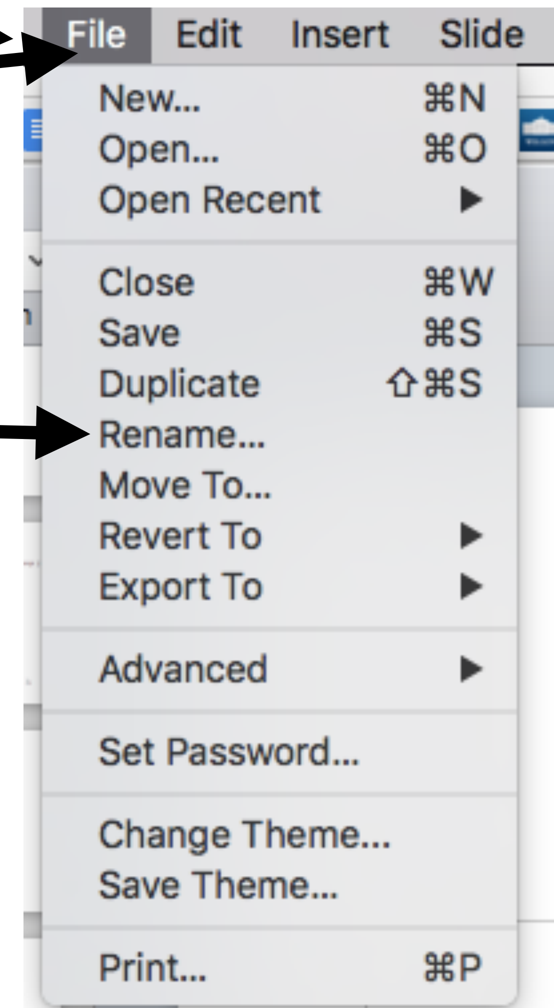
- `JMenuBar`

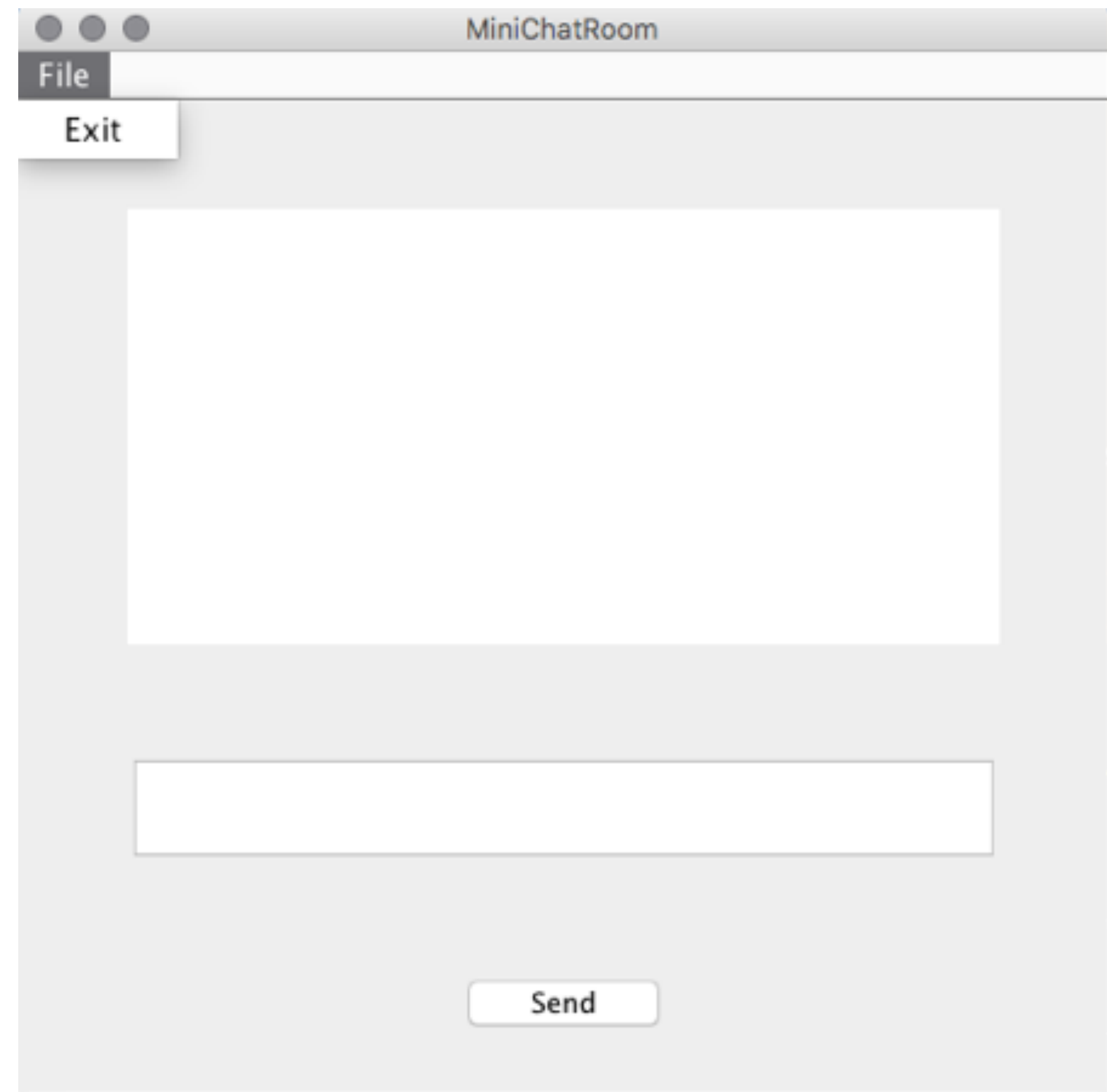
- `JMenu`

- `JMenuItem`

- Per aggiungerle alla finestra:

metodo `void setJMenuBar(JMenuBar)`

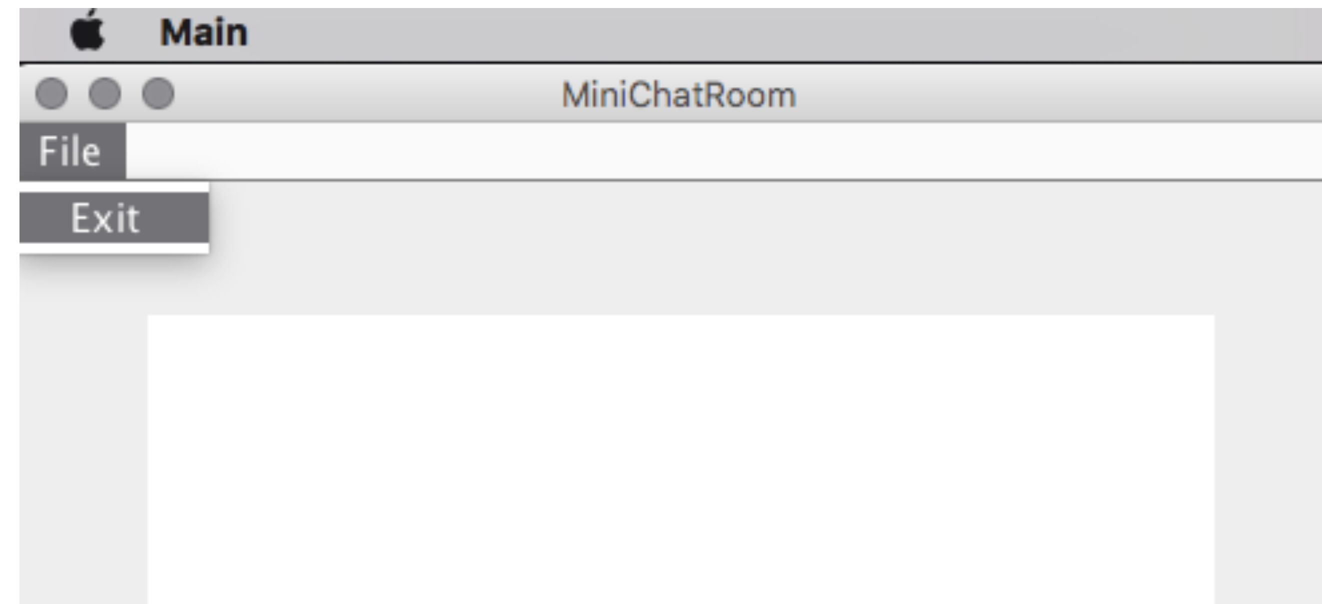


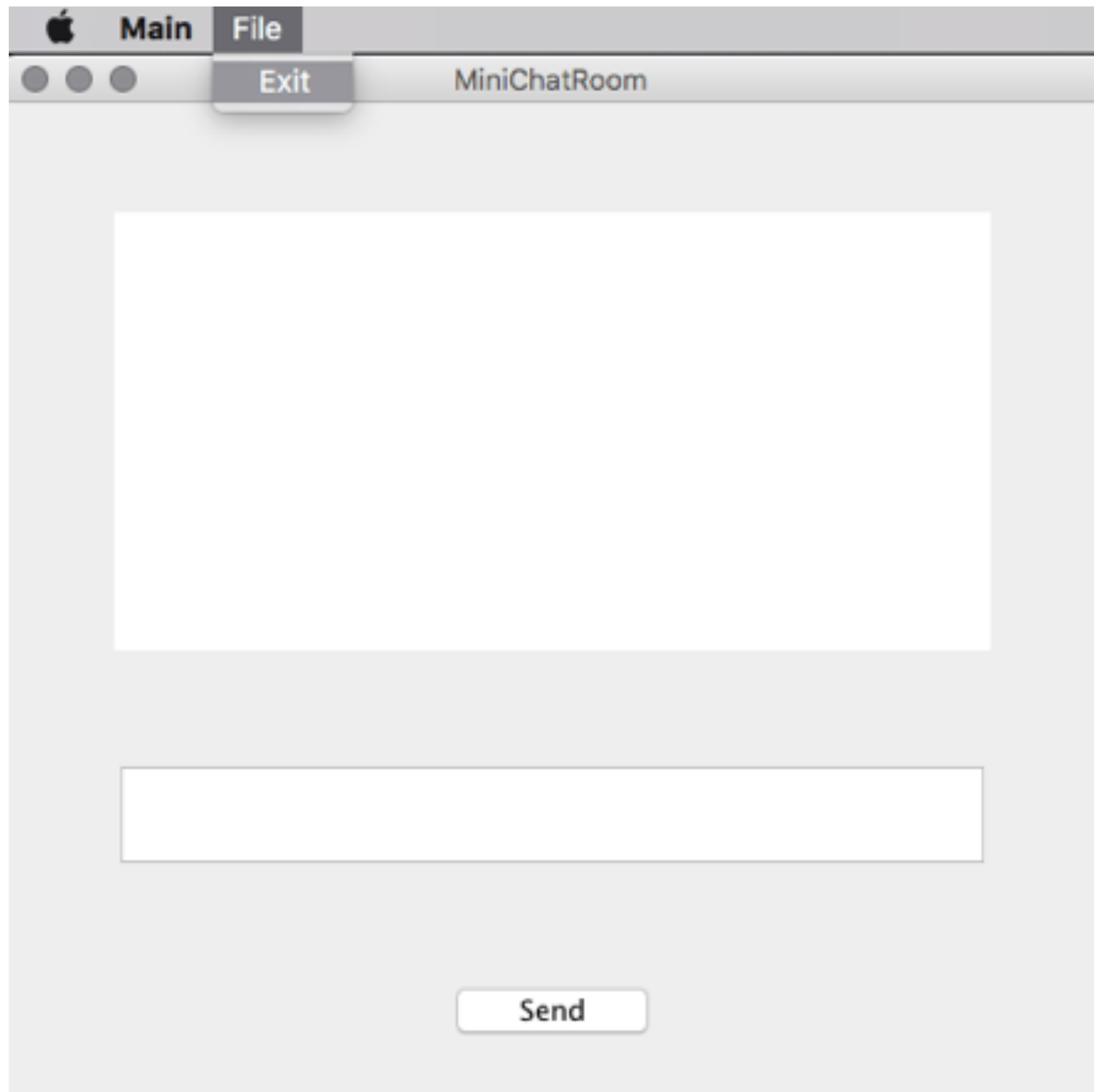


Nel costruttore di ClientGui:

```
JMenuBar menuBar= new JMenuBar();
JMenu chat= new JMenu("File");
menuBar.add(chat);
JMenuItem exit= new JMenuItem("Exit");
chat.add(exit);
this.setJMenuBar(menuBar);
exit.addActionListener(new ExitButtonListener(client));
```

In OS X, i menu sono mostrati in cima allo schermo, non alla finestra





Dobbiamo usare

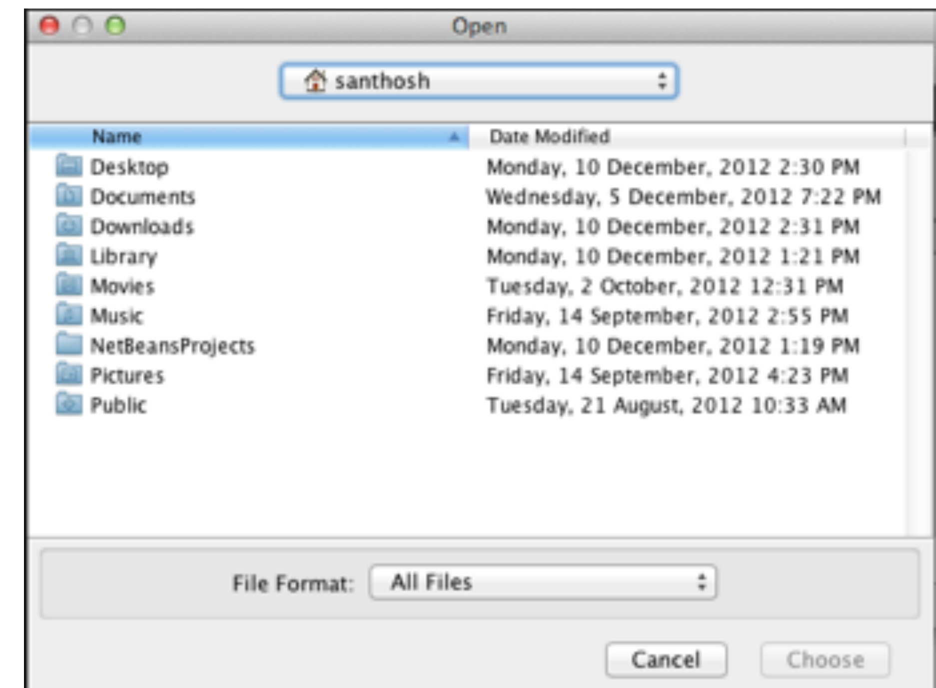
`-Dapple.laf.useScreenMenuBar=true`

Altri componenti

- `JLabel` - testo non clickabile e non editabile sull'interfaccia (esempio nella classe `FatalErrorDialog`)

- `JFileChooser` - per abilitare l'utente di indicare un *file* locale

- `JList` - mostra una lista di elementi che possono essere selezionati dall'utente



- `JScrollPane` - contenitore per componenti "scrollabili"

- `JTabbedPane` - contenitore con tab multipli

- tante altre - consultare documentazione

Posizionare elementi

- Usando posizione assoluta come abbiamo visto
  - Diventa difficile da gestire per applicazioni grandi
- Usando *layout manager*
  - vari tipi di manager
  - impostati usando metodo

```
void setLayout(LayoutManager)
```

I *Panel*

- Contenitori *lightweight*
- Utili per raggruppare componenti
- Classe `JPanel -` Il content pane della finestra è un `JPanel`
- Costruttori:

`JPanel()` - panel con `FlowLayout`

`JPanel(LayoutManager l)` - panel con *layout manager* `l`

- Metodo per per aggiungere dei componenti:

`void add(Component)`

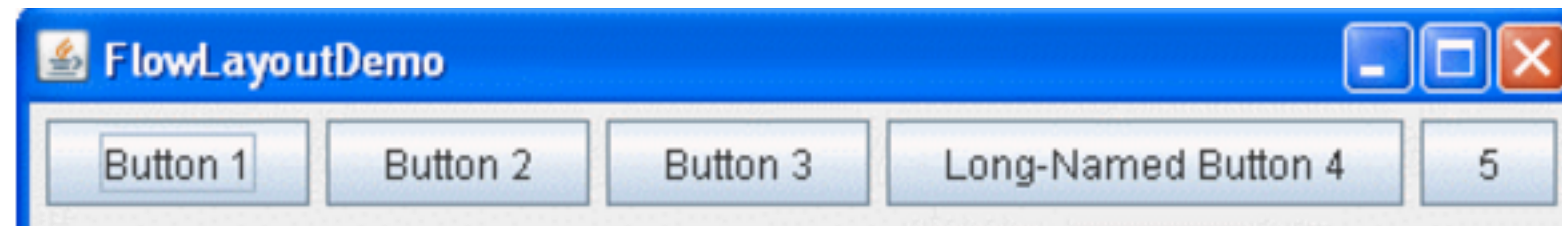
FlowLayout

- Allinea i componenti riga per riga
- Costruttore

```
public FlowLayout(int align, int hgap, int vgap)
```

imposta allineamento (a sinistra, destra, centrale), distanze tra componenti in orizzontale (`hgap`) e verticale (`vgap`)

```
void setHgap(int)

void setVgap(int)
```



BorderLayout

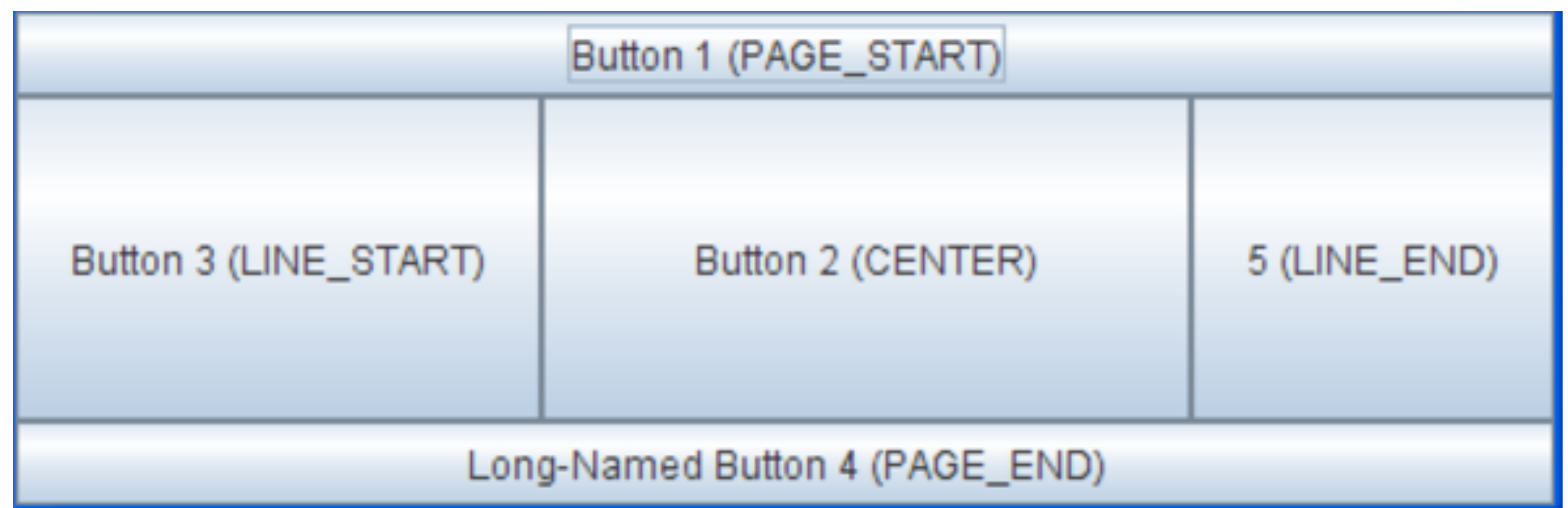
- Divide il panel in 5 aree distinte
- Costruttore:

```
BorderLayout(int horizontalGap, int verticalGap)
```

- Per aggiungere un elemento al panel:

```
void add (Component c, int location)
```

`location` può essere: `BorderLayout.PAGE_START`, `BorderLayout.PAGE_END`, `BorderLayout.LINE_START`, `BorderLayout.LINE_END`, `BorderLayout.CENTER`



GridLayout

- Allinea i componenti in una tabella
- Tutte le celle sono della stessa dimensione
- I componenti prendono la dimensione della cella

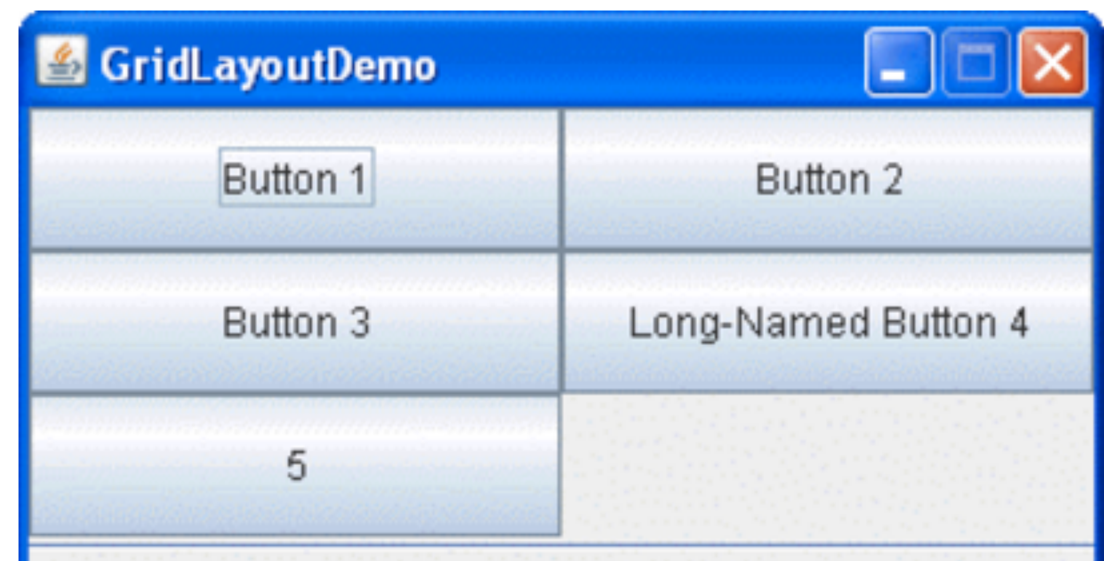
`GridLayout(int rows, int cols, int hgap, int vgap)`

crea un layout con `rows` righe, `cols` colonne, e distanza tra le celle `hgap` e `vgap`.

`void setHgap(int)`

`void setVgap(int)`

impostano distanza tra celle



Interfaccia del progetto

- Più flessibile: non si mescola *input* con *output* come alla riga di comando
- I contenuti che arrivano dal *server* (tramite la *callback*) possono essere mostrati subito
- Le richieste di amicizia possono essere mostrate subito.
- L'interfaccia non è la parte principale del progetto - deve solo facilitare l'interazione con le funzionalità di `SimpleSocial`