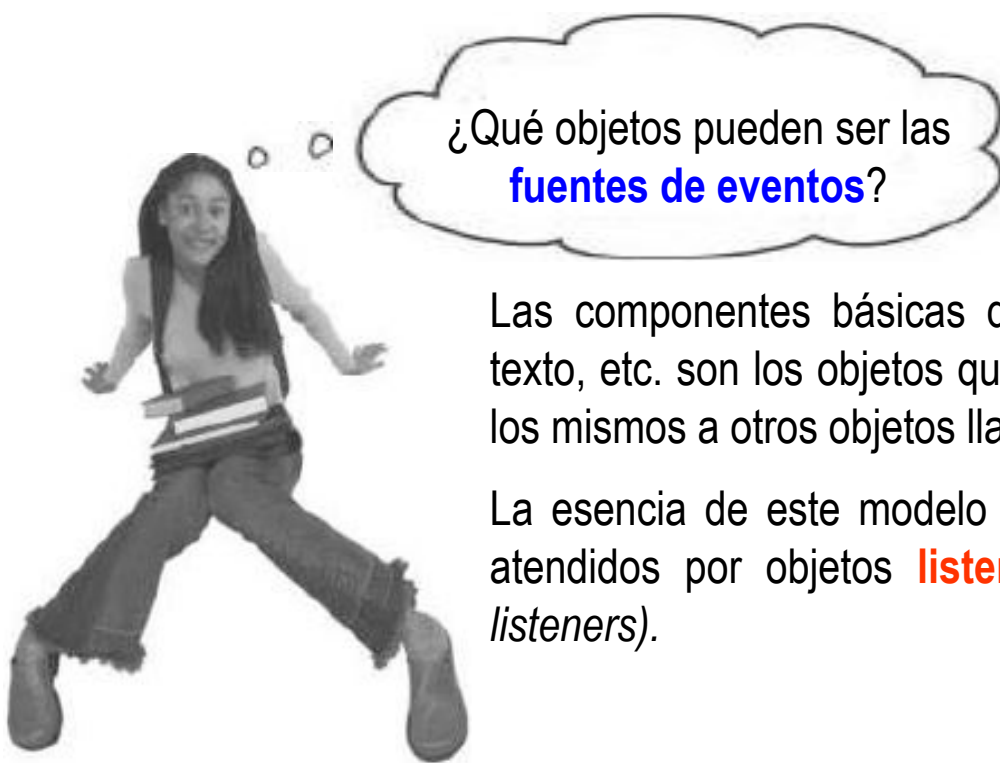


# Manejo de eventos

- Eventos. Jerarquía de clases EventObject
- Generadores de eventos
- Escuchas de eventos (*Listeners*)
  - Implementación de interfaces *listeners*
  - Clases *adapters*
  - Clases Anónimas

# Manejo de eventos

El manejo de eventos de la GUI está basado en el **modelo de delegación**. Dicho modelo se basa en objetos que originan o disparan eventos llamados **fuentes de eventos** y objetos que escuchan y atienden dichos eventos llamados **escuchas de eventos o listeners**.



¿Qué objetos pueden ser las **fuentes de eventos**?

Las componentes básicas de GUI, como botones, listas, campos de texto, etc. son los objetos que disparan eventos y delegan el manejo de los mismos a otros objetos llamados **escuchas**.

La esencia de este modelo es simple: **objetos** que disparan **eventos** atendidos por objetos **listeners** (clases que implementan interfaces *listeners*).

# Manejo de eventos

## Eventos

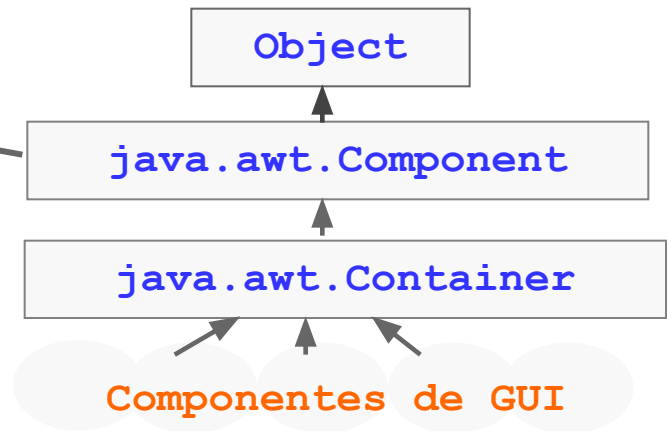
Un **evento** es generado por una componente como consecuencia de una acción iniciada por un usuario (presionar un botón, seleccionar un ítem de una lista, etc.)

## Fuentes de Eventos

Las componentes de GUI, son las que generan eventos. Estas componentes además responden a dos métodos para que los *listeners* registren o quiten interés en los eventos que ellas generan:

```
public void addXXXListener(XXXListener)
public void removeXXXListener(XXXListener)
```

Cuando una componente genera un evento, éste es pasado al manejador o a los manejadores que se registraron en ella.



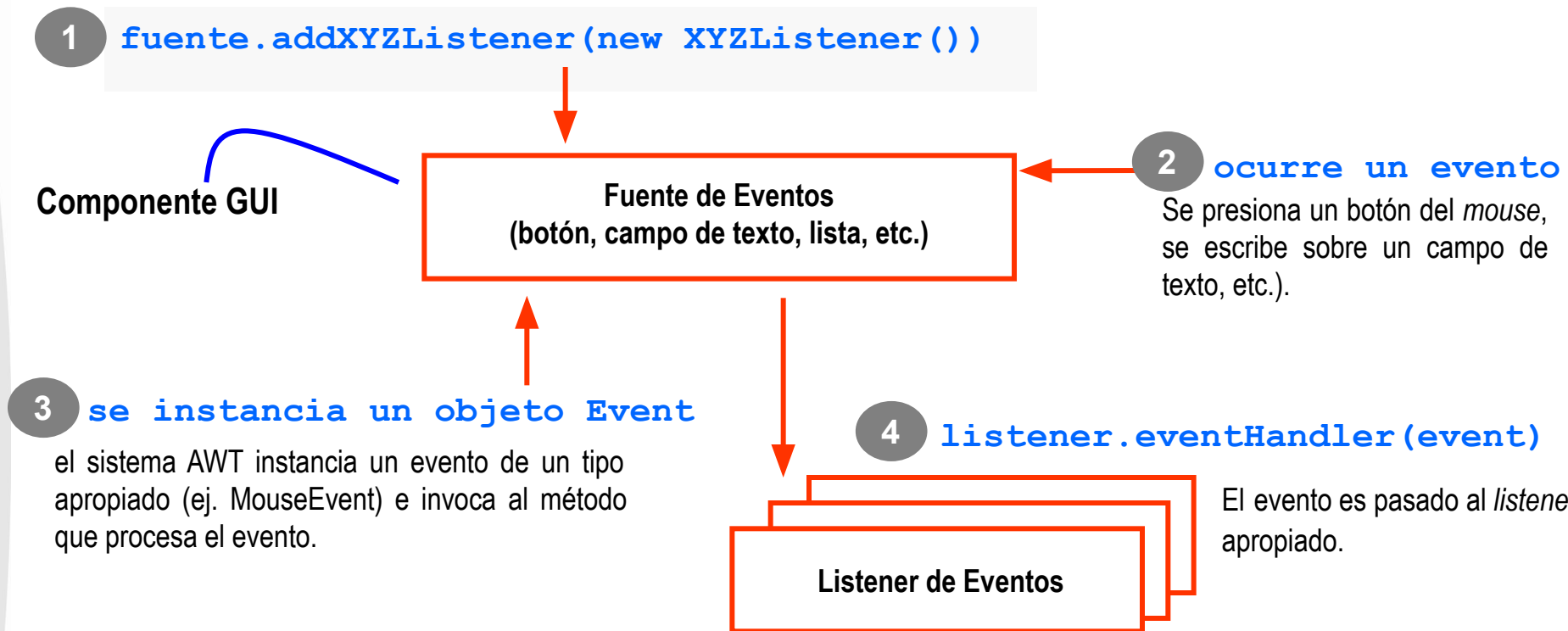
## Manejadores de eventos (Listeners)

Un *listener* es un objeto que implementa una determinada interface. Los métodos de estos objetos, reciben como parámetro un **AWTEvent** específico, que contiene información sobre el evento y sobre la componente AWT que lo disparó.

# Manejo de eventos

Sobre una componente de interfaz de usuario, se pueden registrar uno o más *listeners*. El orden en que los *listeners* son notificados del evento, es indefinido.

## ¿Cómo funciona la delegación de eventos?



# Interfaces EventListener en AWT

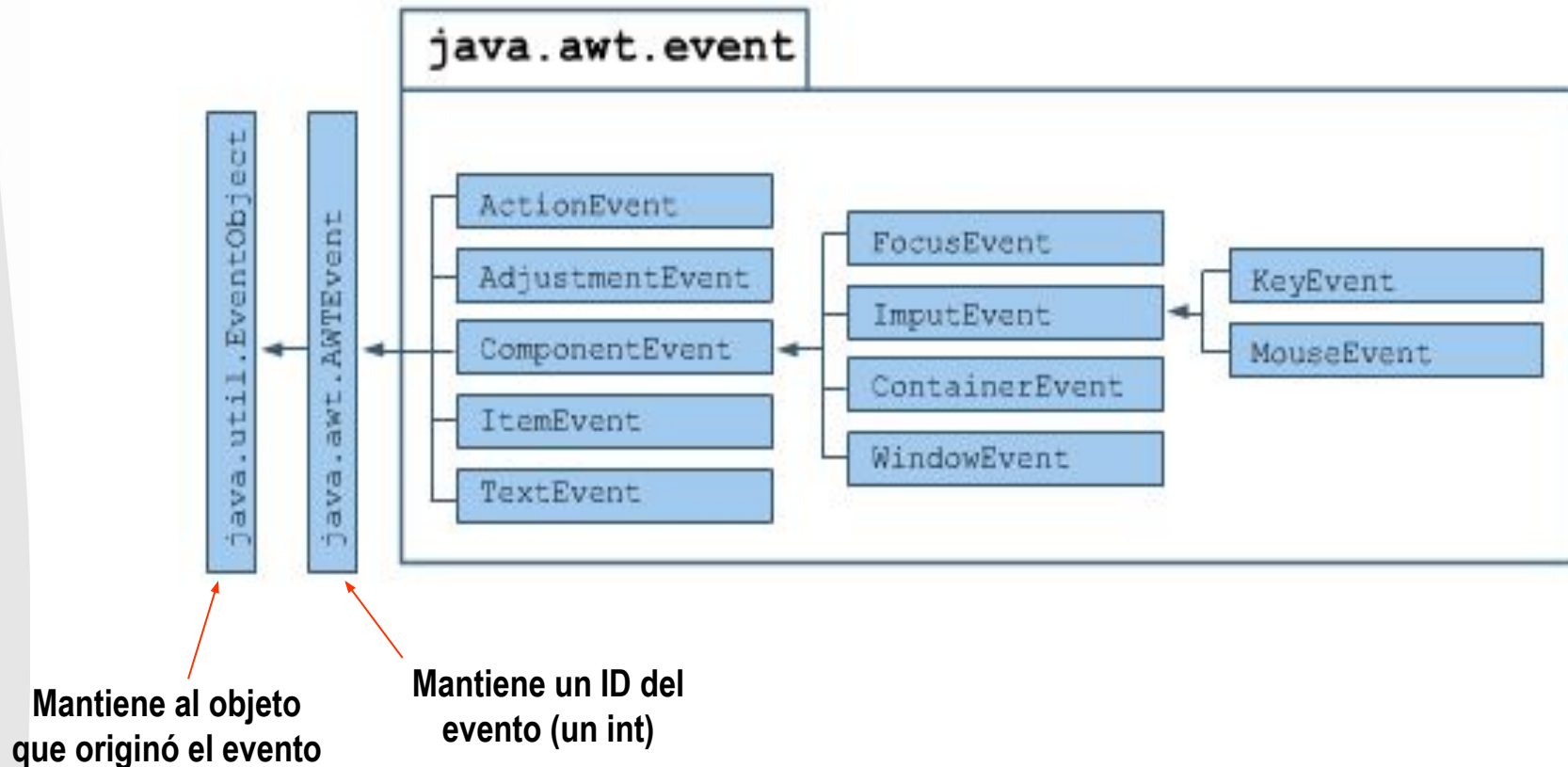
Para cada categoría de eventos, hay una interface que debe ser implementada. Cada interface tiene uno o más métodos que deben ser implementados y serán invocados cuando ocurre un evento específico sobre la componente. Todas son subclases de `java.util.EventListener`

Interface o clase que la implementa	Métodos de la interface
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
ComponentListener ComponentAdapter	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
KeyListener KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)

Interface o clase que la implementa	Métodos de la interface
MouseListener MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)
MouseWheelListener	mouseWheelMoved(MouseWheelEvent e)

# Tipos de eventos en AWT

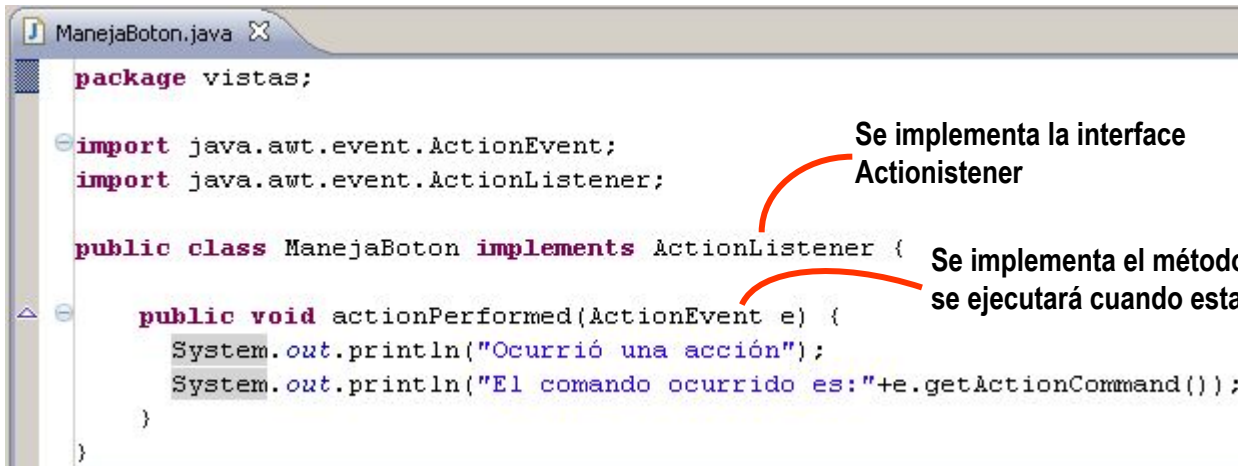
Como vimos en los métodos de las interfaces, hay distintas categorías de eventos. Hay una jerarquía de eventos, que es la siguiente:



La clase `java.util.EventObject` junto con las interfaces analizadas, constituyen el fundamento del modelo de delegación de eventos.

# Manejo de eventos

Para crear un objeto *listener*, se debe implementar alguna de las interfaces *listener* provistas por la API. En este ejemplo se implementa la interface `ActionListener`, que maneja eventos genéricos de tipo `ActionEvent` y que tiene sólo el método `actionPerformed(ActionEvent e)`.



```
package vistas;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class ManejaBoton implements ActionListener {

    public void actionPerformed(ActionEvent e) {
        System.out.println("Ocurrió una acción");
        System.out.println("El comando ocurrido es:" + e.getActionCommand());
    }
}
```

Se implementa la interface `ActionListener`

Se implementa el método `actionPerformed()` que se ejecutará cuando esta clase sea notificada.

¿Qué sucede si se presiona un botón que tiene registrado este manejador?

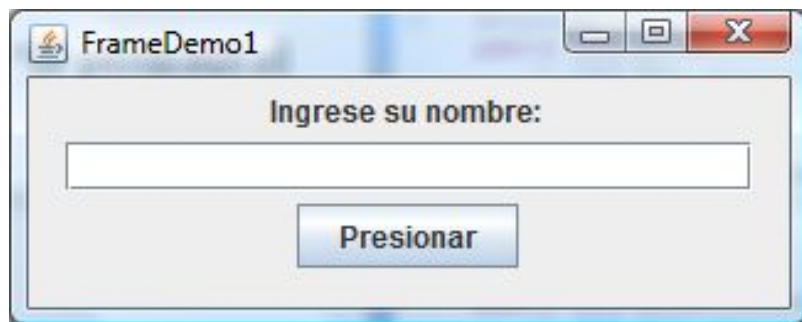
Se imprime en la consola lo siguiente:

```
Ocurrió una acción
El comando del botón es: ButtonPressed
```



# Manejo de eventos – Un Ejemplo

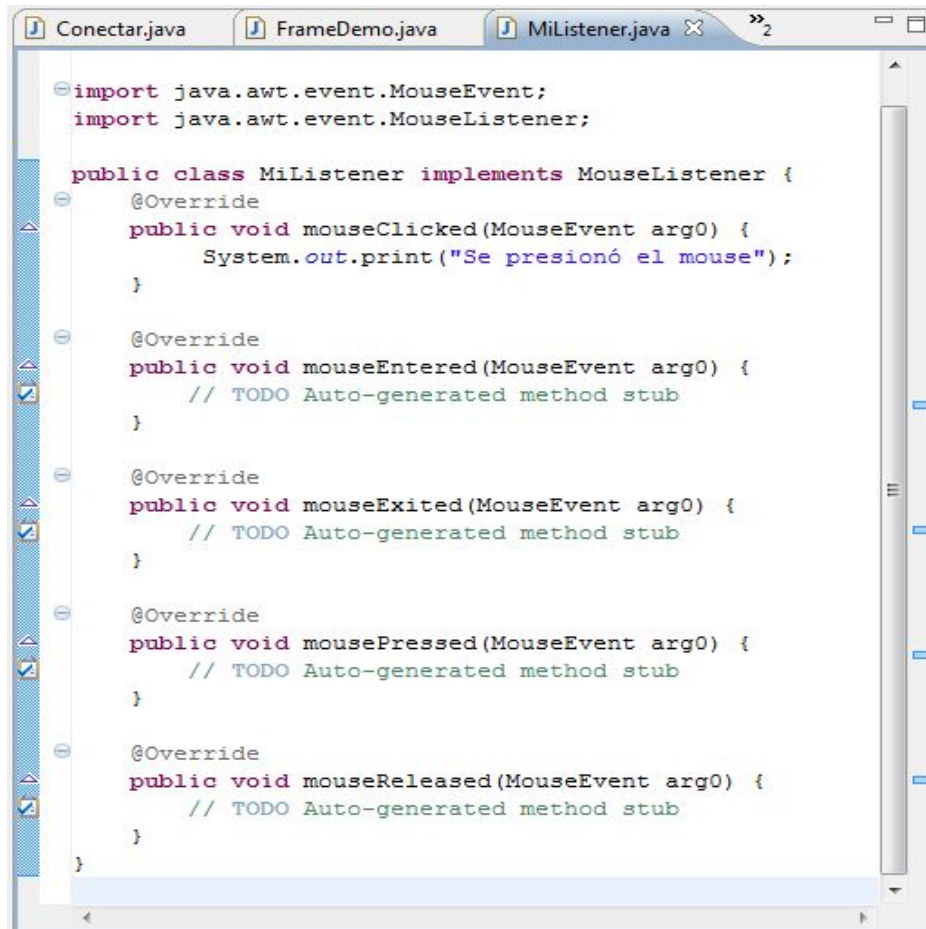
Supongamos que tenemos la siguiente clase, llamada **MiListener** que implementa la interface **MouseListener** y está interesada sólo en saber cuando se hace “click” del mouse sobre el botón. Supongamos que se registra este listener en el botón “Presionar” ¿Qué sucede al presionarlo?



Se imprime en la consola “**Se presionó el mouse**”.  
¿Qué sucede cuando me posiciono sobre el botón?

No hace NADA!!

Sólo interesa el método **mouseClicked(MouseEvent arg0)**, sin embargo se debieron implementar TODOS los métodos y dejar sus cuerpos vacíos.



¿Hay otra opción? **Si Clases Adapters**



# Clases Adapter

Para no tener que implementar todos los métodos definidos en la interface, AWT provee un conjunto de clases que implementan las interfaces listeners. Estas clases llamadas Adapters dejan todos los cuerpos de los métodos vacíos.

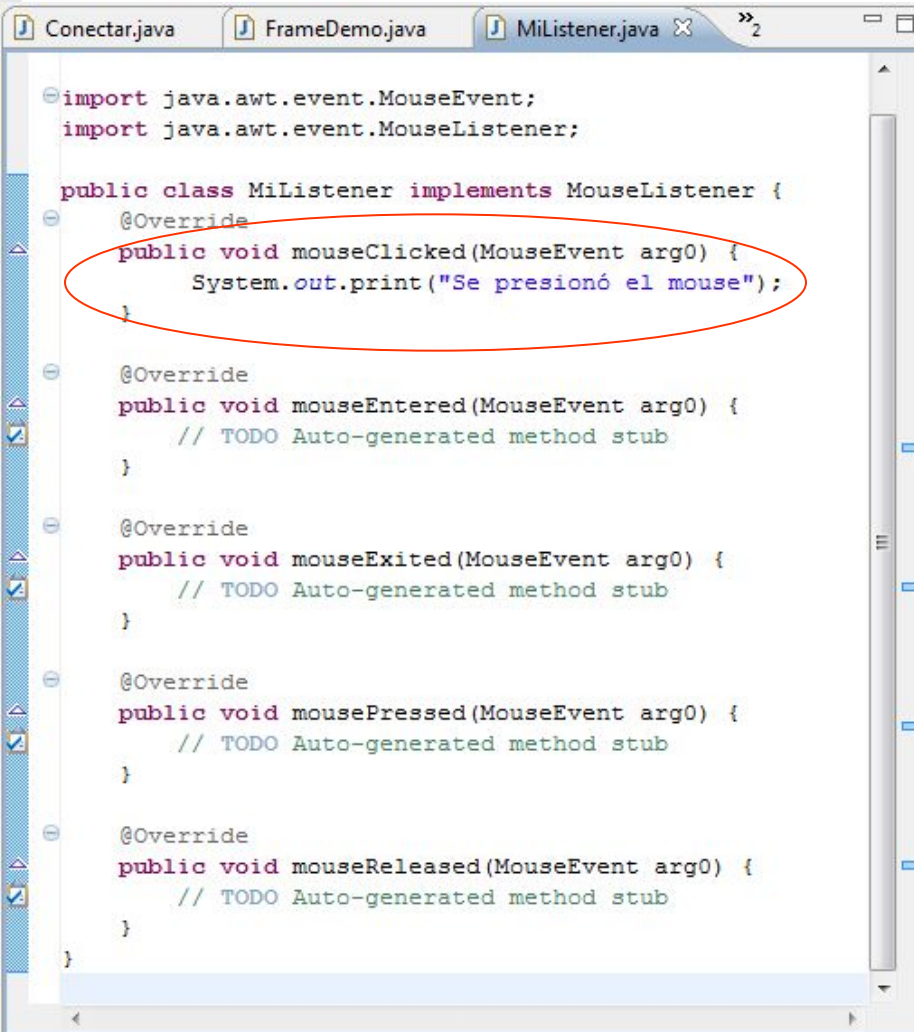
Por lo tanto, para crear un Listener, se puede extender una clase adaptadora y sobrescribir solamente el/los método/s que interesan.

<i>Interface listener ó Clase que la implementa</i>	<i>Métodos de la interface</i>
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
ComponentListener ComponentAdapter	componentHidden(ComponentEvent) componentShown(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent)
ContainerListener ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
KeyListener KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)

<i>Interface listener ó Clase que la implementa</i>	<i>Métodos de la interface</i>
MouseListener MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)
MouseMotionListener MouseMotionAdapter	mouseDragged(MouseEvent) mouseMoved(MouseEvent)
WindowListener WindowAdapter	windowOpened(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent) windowActivated(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)
MouseWheelListener	mouseWheelMoved(MouseWheelEvent e)

# Clases Adapter – Un ejemplo

Si se modifica el ejemplo anterior para usar la clase **Mousedapter**, el código quedaría así:



```
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

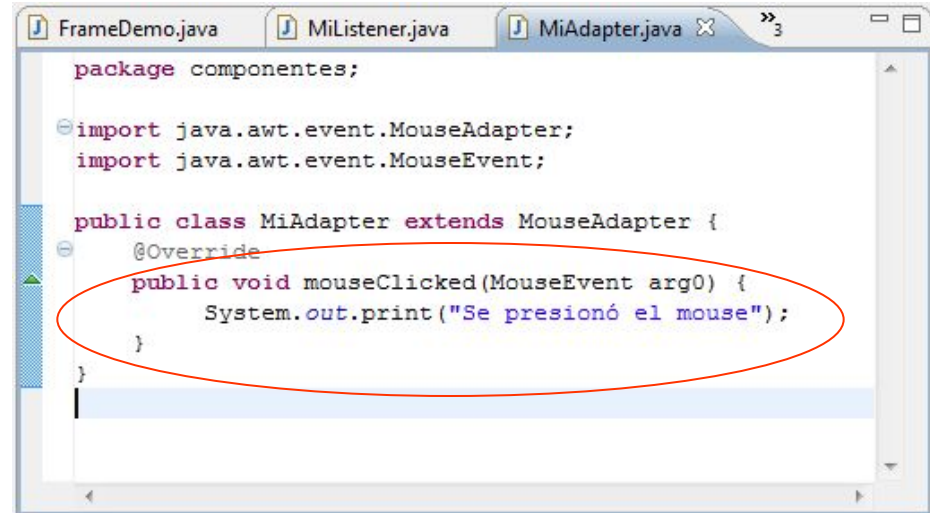
public class MiListener implements MouseListener {
    @Override
    public void mouseClicked(MouseEvent arg0) {
        System.out.print("Se presionó el mouse");
    }

    @Override
    public void mouseEntered(MouseEvent arg0) {
        // TODO Auto-generated method stub
    }

    @Override
    public void mouseExited(MouseEvent arg0) {
        // TODO Auto-generated method stub
    }

    @Override
    public void mousePressed(MouseEvent arg0) {
        // TODO Auto-generated method stub
    }

    @Override
    public void mouseReleased(MouseEvent arg0) {
        // TODO Auto-generated method stub
    }
}
```



```
package componentes;

import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;

public class MiAdapter extends MouseAdapter {
    @Override
    public void mouseClicked(MouseEvent arg0) {
        System.out.print("Se presionó el mouse");
    }
}
```

Usando clases adaptadoras, la clase se simplifica notablemente.

# Registrando Listeners



Creamos componentes, analizamos listeners y eventos, ahora ...  
¿Cómo registro listeners en componentes?

Para registrar un *listener* en una componente se utiliza el método **addxxxListener(unListener)**, donde xxx es la categoría de evento (mouse, window, key, etc.)

Continuando con el ejemplo anterior, se quiere registrar interés en los eventos del *mouse* producidos por un botón, entonces deberíamos poner:

```
unBoton.addMouseListener(unListener)
```

Instancia de una clase que implementó `MouseListener`

# Registrando Listeners

La registración usando la clase **MiListener** que implementa la interface **MouseListener** o la clase **MiAdapter** que extiende la clase **MouseAdapter** es idéntica:

```
package componentes;

import java.awt.Dimension;

public class FrameDemo1 extends JFrame {
    private JLabel label = new JLabel("Ingrese su nombre: ");
    private JTextField text = new JTextField(25);
    private JButton button = new JButton("Presionar");

    private FrameDemo1() {
        super("FrameDemo1");
        //Create a panel and add components to it.
        JPanel contentPane = new JPanel(new FlowLayout());
        contentPane.add(label);
        contentPane.add(text);
        contentPane.add(button);
        button.addMouseListener(new MiAdapter());

        this.setPreferredSize(new Dimension(200, 100));
        this.getContentPane().add(contentPane);
    }

    public static void main(String[] args) {
        FrameDemo1 frame = new FrameDemo1();
        // Display the window.
        frame.pack();
        frame.setVisible(true);
    }
}
```





# Listeners de Eventos

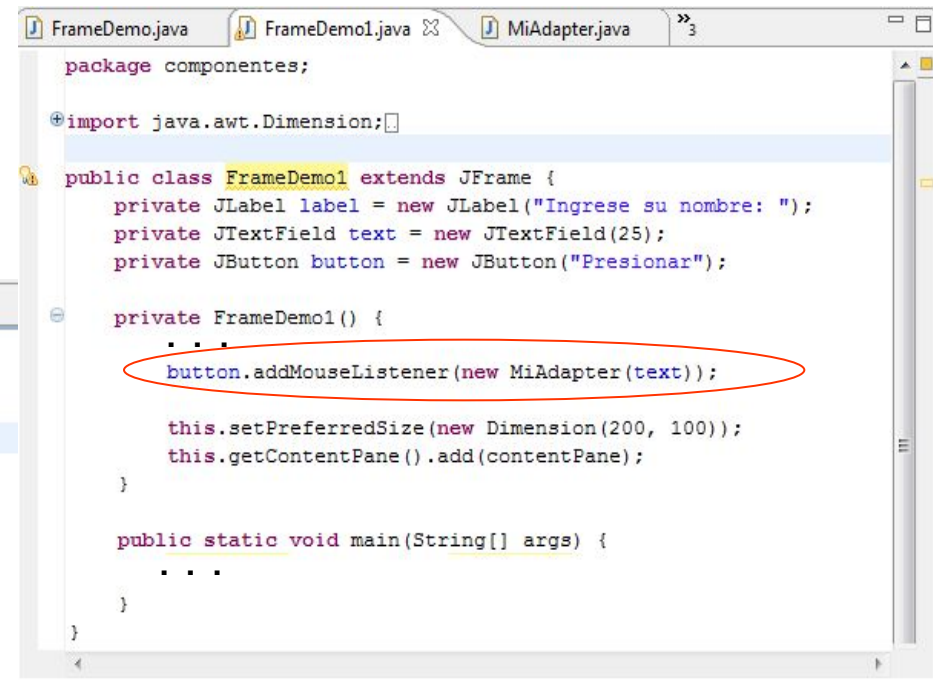
**Esto no es recomendable!!!**

¿Dónde se ubican las clases listeners?

En el ejemplo anterior la clase de la GUI es independiente de las clases que manejan eventos (lo único que hace la clase **MiListener** es imprimir en la consola).

¿Qué pasa si los listener necesitan acceder a las componentes de la GUI?

Supongamos que el listener **MiAdapter** quiere acceder a la variable de instancia **text**.



```
package componentes;

import java.awt.event.MouseAdapter;

public class MiAdapter extends MouseAdapter {
    private JTextField text = null;
    public MiAdapter(JTextField text) {
        this.text = text;
    }

    @Override
    public void mouseClicked(MouseEvent arg0) {
        System.out.print("Se presionó el mouse "+text.getText());
    }
}

package componentes;

import java.awt.Dimension;

public class FrameDemo1 extends JFrame {
    private JLabel label = new JLabel("Ingrese su nombre: ");
    private JTextField text = new JTextField(25);
    private JButton button = new JButton("Presionar");

    private FrameDemo1() {
        button.addMouseListener(new MiAdapter(text));

        this.setPreferredSize(new Dimension(200, 100));
        this.getContentPane().add(contentPane);
    }

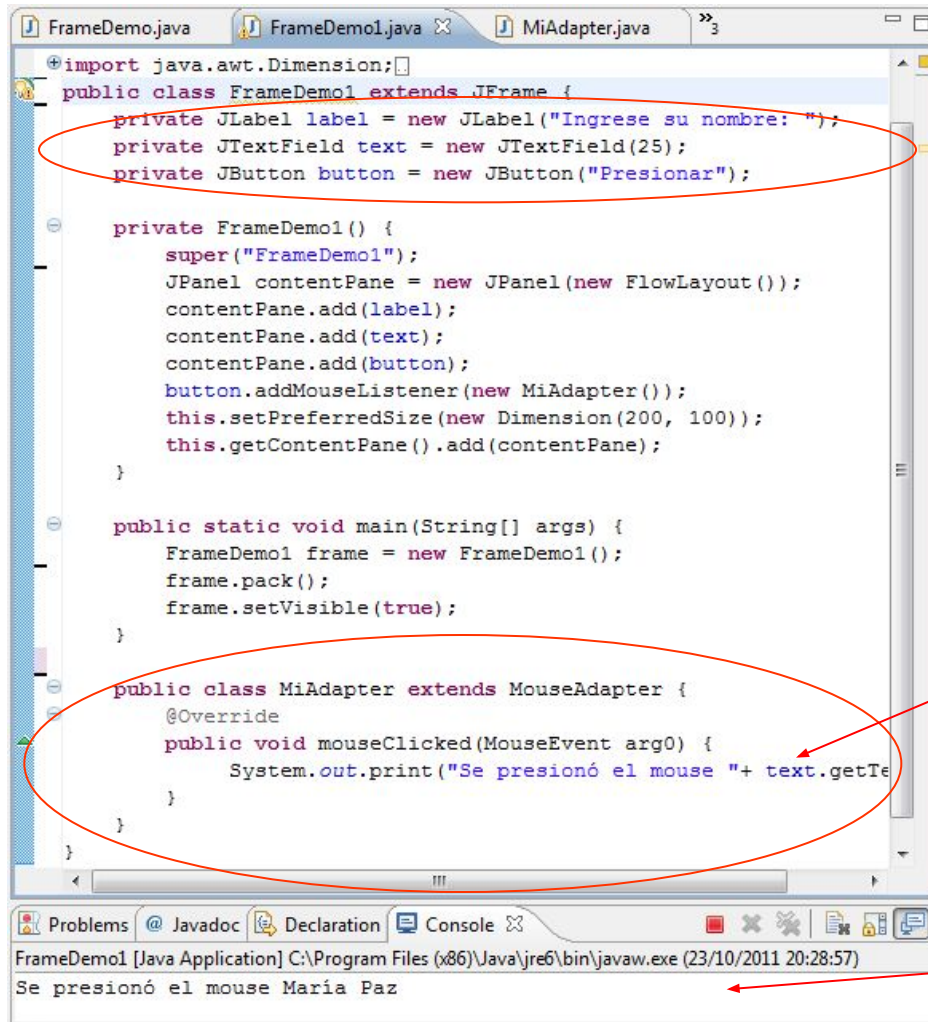
    public static void main(String[] args) {
        . . .
    }
}
```

**Si tenemos que pasar todos los campos se hace muy complicado!!** Para evitar esto, se suele definir a las clases *listener* dentro de la clase que modela a la GUI, como clases internas.

# Listeners de Eventos

## Clases internas

Se recomienda escribir la clase **MiAdapter** o **MiListener** directamente adentro de la clase que define la Interfaz de Usuario Gráfica o Vista de la aplicación.



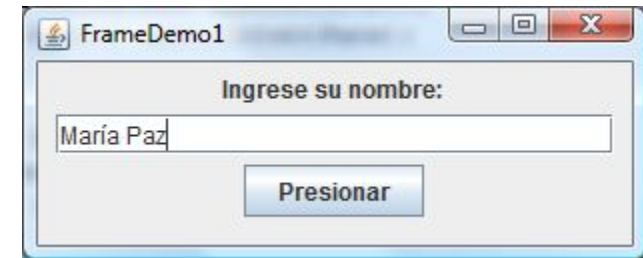
```
import java.awt.Dimension;

public class FrameDemo1 extends JFrame {
    private JLabel label = new JLabel("Ingrese su nombre: ");
    private JTextField text = new JTextField(25);
    private JButton button = new JButton("Presionar");

    private FrameDemo1() {
        super("FrameDemo1");
        JPanel contentPane = new JPanel(new FlowLayout());
        contentPane.add(label);
        contentPane.add(text);
        contentPane.add(button);
        button.addMouseListener(new MiAdapter());
        this.setPreferredSize(new Dimension(200, 100));
        this.getContentPane().add(contentPane);
    }

    public static void main(String[] args) {
        FrameDemo1 frame = new FrameDemo1();
        frame.pack();
        frame.setVisible(true);
    }

    public class MiAdapter extends MouseAdapter {
        @Override
        public void mouseClicked(MouseEvent arg0) {
            System.out.print("Se presionó el mouse " + text.getText());
        }
    }
}
```



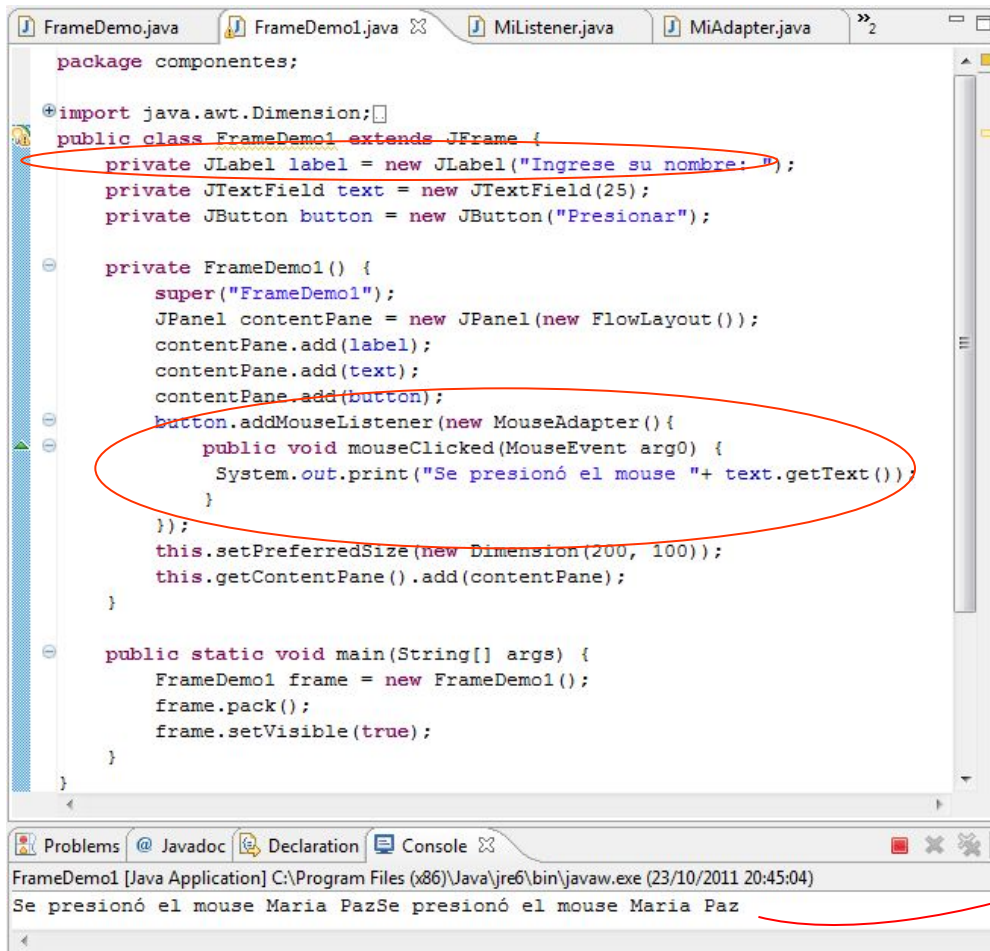
Las clases internas tienen acceso a los miembros de la clase que las contiene.

Al hacer doble click sobre el botón **Presionar**, se imprime en la consola

# Listeners de Eventos

## Clases anónimas

Otra alternativa para manejar los eventos es utilizar clases anónimas. Las clases anónimas son clases internas sin nombre, vinculadas a un único objeto, ideales para manejar eventos generados por un único objeto de interfaz de usuario.



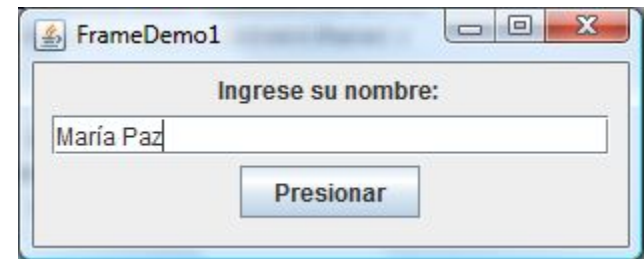
```
package componentes;

import java.awt.Dimension;

public class FrameDemo1 extends JFrame {
    private JLabel label = new JLabel("Ingrese su nombre: ");
    private JTextField text = new JTextField(25);
    private JButton button = new JButton("Presionar");

    private FrameDemo1() {
        super("FrameDemo1");
        JPanel contentPane = new JPanel(new FlowLayout());
        contentPane.add(label);
        contentPane.add(text);
        contentPane.add(button);
        button.addMouseListener(new MouseAdapter() {
            public void mouseClicked(MouseEvent arg0) {
                System.out.print("Se presionó el mouse " + text.getText());
            }
        });
        this.setPreferredSize(new Dimension(200, 100));
        this.getContentPane().add(contentPane);
    }

    public static void main(String[] args) {
        FrameDemo1 frame = new FrameDemo1();
        frame.pack();
        frame.setVisible(true);
    }
}
```



Al hacer doble click sobre el botón **Presionar**, se imprime en la consola



# Listeners de Eventos

## Eventos del Teclado

Los eventos de teclado (*key events*) son disparados por algunas componentes de GUI cuando el usuario presiona o libera una tecla del teclado. Las notificaciones son acerca de dos clases de eventos:

- La tipificación de un carácter Unicode (*key pressed*)
- El evento de presionar o liberar alguna tecla (*key-pressed or key-released*)

```
package juego;  
import java.awt.event.KeyEvent;  
import java.awt.event.KeyListener;
```

Se implementa la interface  
KeyListener

```
...  
public class MoverConTeclado implements KeyListener {
```

```
public void keyReleased(KeyEvent evt) {  
}
```

```
public void keyPressed(KeyEvent evt) {
```

Se implementa el método keyPressed() para  
analizar la tecla presionada

```
    int ckey = evt.getKeyCode();  
    Point p = pac.getLocation();  
    if (ckey == 38) { //Up  
        pac.setLocation(new Point((int)p.getX(), (int)p.getY()-10));  
        pac.setIcon(pacimg[3]);  
    }
```

```
    if (ckey == 40) { //Down  
        pac.setLocation(new Point((int)p.getX(), (int)p.getY()+10));  
        pac.setIcon(pacimg[2]);  
    }
```

```
    ...
```

```
public void keyTyped(KeyEvent evt) {
```

```
}
```

```
...  
}
```

Métodos de la interface

# Listeners de Eventos

## Eventos del Teclado

```
package juego;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
. . .
public class MoverConTeclado {
    public ImageIcon[] pacimg = new ImageIcon[4];
    private JLabel pac = null;

    private void createAndShowGUI() {
        JPanel gamePanel = new JPanel();
        this.LoadGraphics();
        pac = new JLabel(pacimg[0]);
        pac.setLocation(50,50);
        pac.setFocusable(true);
        pac.addKeyListener(new ManejaEventosTeclado());
        gamePanel.add(pac);
        frame = new JFrame("PacMan ALONE");
        frame.setPreferredSize(new Dimension(300, 500));
        frame.getContentPane().add(gamePanel);
        frame.pack();
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        MoverConTeclado pacman = new MoverConTeclado();
        pacman.createAndShowGUI();
    }
    . . .
}
```

Para que una componente dispare eventos del teclado, debe tener el foco del teclado

En este caso se extiende la clase KeyAdapter y es una clase interna de MoverConTeclado

```
class ManejaEventosTeclado extends KeyAdapter {
    public void keyPressed(KeyEvent evt) {
        int ckey = evt.getKeyCode();
        Point p = pac.getLocation();
        if (ckey == 38) {
            pac.setLocation(
                new Point((int) p.getX(), (int) p.getY() - 10));
            pac.setIcon(pacimg[3]);
            NewKeydir = 1; // UP
        }
        if (ckey == 40) {
            pac.setLocation(
                new Point((int) p.getX(), (int) p.getY() + 10));
            pac.setIcon(pacimg[2]);
            NewKeydir = 3; // Down
        }
        . . .
    }
}
```