ActionListener

El método `addActionListener()` permite registrar un objeto que "escucha" eventos y ejecuta código cuando se produce la acción.

Un ActionListener es una interfaz funcional (tiene un solo método abstracto):

```
public interface ActionListener extends EventListener {
    void actionPerformed(ActionEvent e);
}
```

Cada vez que ocurre una acción (click en un botón), se llama a `actionPerformed`.

¿Cómo usar addActionListener?

1. Clase anónima

```
import javax.swing.*;
import java.awt.event.*;

public class acciones{
    public static void main(String[] args) {
        JFrame frame = new JFrame("Ejemplo addActionListener");
        JButton boton = new JButton("Haz clic");

        boton.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                System.out.println("¡Botón presionado!");
            }
        });

        frame.add(boton);
        frame.setSize(300, 200);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```

Aquí usamos una clase anónima y aplicamos `@Override` porque estamos sobrescribiendo el método actionPerformed de la interfaz.

2. Lambda (desde Java 8)

Como ActionListener es una interfaz funcional, podemos usar expresiones lambda:

```
boton.addActionListener(e -> System.out.println("Clic con
lambda!"));
```

Aquí no hace falta `@Override`, porque no estamos escribiendo una clase, solo implementamos la interfaz con una expresión más corta.

3. Implementando la interfaz en la clase

Podemos hacer que la propia clase implemente ActionListener:

```
import javax.swing.*;
import java.awt.event.*;

public class Ejemplo3 extends JFrame implements ActionListener
{
    JButton boton;

    public Ejemplo3() {
        boton = new JButton("Click aquí");
        boton.addActionListener(this);
        add(boton);

        setSize(300, 200);
        setDefaultCloseOperation(EXIT_ON_CLOSE);
        setVisible(true);
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Botón presionado desde la clase
principal");
    }

    public static void main(String[] args) {
        new Ejemplo3();
    }
}
```

Aquí el @Override es obligatorio porque implementamos la interfaz directamente en la clase.

4. Múltiples botones con un solo listener

Podemos usar el mismo listener para varios componentes:

```
JButton boton1 = new JButton("Aceptar");
JButton boton2 = new JButton("Cancelar");

ActionListener listener = new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == boton1) {
            System.out.println("Se presionó Aceptar");
        } else if (e.getSource() == boton2) {
            System.out.println("Se presionó Cancelar");
        }
    }
};

boton1.addActionListener(listener);
boton2.addActionListener(listener);
```

Entonces… ¿Cuándo usar @Override?

Cuando implementas una interfaz o sobrescribes un método heredado.
Ejemplo:

```
 @Override
public void actionPerformed(ActionEvent e) { ... }
```

- No se usa con lambdas, porque ahí no escribes un método explícito, solo pasas una implementación.
- Es recomendable usarlo siempre que sobreescribas un método, porque:
    - Evita errores .
    - Hace más claro que estás implementando un método de la interfaz.

Consejos:

- Usar **lambda** si el código de la acción es corto y legible.
- Usar **clase anónima** si es necesario más líneas o acceso a variables locales.
- **Implementar la interfaz en la clase** solo si muchos componentes usan el mismo ActionListener o si se desea centralizar la lógica.