Interfaces gráficas de usuario Swing: Componentes básicos

Introducción

Existe una gran cantidad de componentes Swing, puedes echarles un vistazo en Guía de componentes Swing.

En este capítulo, vamos a estudiar cómo usar algunos de ellos.

Empezaremos por los más sencillos, e iremos avanzando hasta llegar a ver componentes muy sofisticados.

Bibliografía

1. Big Java. Capítulo 17.
2. Head first Java. Capítulo 12.
3. Head first design patterns. Capítulo 12.
4. Desarrollo de proyectos informáticos con tecnología Java. Capítulo 11.

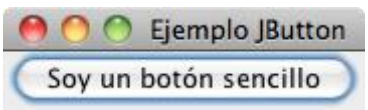
Contenidos

1. Botones.
    1. Sin estado.
    2. Con estado.
    3. Con estado y excluyentes.
2. Mostrar texto.
    1. Que no se puede modificar.
    2. En una única línea.
    3. En párrafos.
3. Barras de desplazamiento -scroll-
4. Listas.
5. Sliders.

Botones sin estado

Un botón sin estado es aquel sobre el que podemos hacer *click* pero que no queda pulsado después de ello.

La clase **JButton** define un componente con este tipo de comportamiento.



```
JFrame ventana = new JFrame("Ejemplo JButton");
Container contenedor = ventana.getContentPane();
JButton boton = new JButton("Soy un botón sencillo");
contenedor.add(boton);
ventana.pack();
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
ventana.setVisible(true);
```

El evento de interés más común que este componente genera es **ActionEvent**, que ya sabemos que podemos escuchar implementando la **interface ActionListener**.

```
boton.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Me pulsaste");
    }
});
```

Ahora, cada vez que se pulse el botón, se mostrará un mensaje por consola.

Podemos tener el mismo escuchador para más de un botón:

```
JButton boton = new JButton("Soy un botón sencillo");
ActionListener escuchador;
boton.addActionListener(escuchador = new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Me pulsaste.");
    }
});
JButton otroBoton = new JButton("Soy otro botón");
otroBoton.addActionListener(escuchador);
```

¿Cómo podemos saber qué botón generó el evento?

Tenemos, al menos, dos opciones:

1. Preguntar al evento por el componente que lo generó:

   ```
   public void actionPerformed(ActionEvent e) {
       JButton boton = (JButton)e.getSource(); //Devuelve Object
       ...
   }
   ```

2. Asignar un *command* al botón:

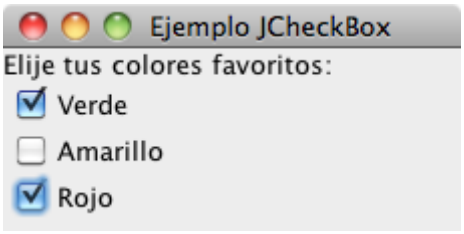
   ```
   boton.setActionCommand("comando1");
   ...
   public void actionPerformed(ActionEvent e) {
       String comando = e.getActionCommand();
       ...
   }
   ```

Si asignamos un *command* a cada uno de los botones, gracias a que la versión Java 1.7 permite hacer **switch** sobre **String** podremos filtrar rápidamente el botón que fue pulsado.

```
switch(e.getActionCommand()) {
    case "comando1":
        //Código de respuesta
        break;
    case "comando2":
        //Código de respuesta
        break;
}
```

Si necesitas botones que guarden su estado, para saber si están pulsados o no, puedes usar las clases: **JCheckBox** o **JRadioButton**.

Veamos primero cómo funcionan los **JCheckBox**.



Como puedes ver, si necesitas poder seleccionar más de un botón, utiliza **JCheckBox**.

El código del ejemplo anterior:

```
JFrame ventana = new JFrame("Ejemplo JCheckBox");
JCheckBox rojo = new JCheckBox("Rojo");
JCheckBox amarillo = new JCheckBox("Amarillo");
JCheckBox verde = new JCheckBox("Verde");
JPanel colores = new JPanel();
colores.setLayout(new BoxLayout(colores, BoxLayout.PAGE_AXIS));
ventana.setContentPane(colores);//Cambiamos el panel contenedor
colores.add(new JLabel("Elije tus colores favoritos:"));
colores.add(verde);
colores.add(amarillo);
colores.add(rojo);
ventana.pack();
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
ventana.setVisible(true);
```

Si buscamos los escuchadores que le podemos añadir a un **JCheckBox** veremos que son los mismos que los que podemos añadir a un **JButton**.

Veamos primero el comportamiento al usar **ActionListener**.

```
rojo.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Botón pulsado.");
    }
});
```

Con este escuchador, cada vez que pulsemos el botón obtendremos un mensaje en consola.

Botones con estado

Si necesitamos conocer si lo que ha ocurrido es que el botón ha quedado pulsado tras el *click*, o no ha quedado pulsado, necesitamos un evento que nos dé más información que **ActionEvent**.

Ese evento es **ItemEvent**, que podemos prever que lo podemos escuchar implementando la **interface ItemListener**.

Recuerda que, quien lleva una descripción sobre lo que ha ocurrido es la referencia a **ItemEvent**, a ella le preguntaremos por la información que queremos conocer.

Recuerda, también, que Swing utiliza el método *push* para notificar a sus escuchadores.

ItemListener declara sólo **itemStateChanged(ItemEvent e)**.

```
rojo.addItemListener(new ItemListener() {
    @Override
    public void itemStateChanged(ItemEvent e) {
        switch(e.getStateChange()) {//Preguntamos al evento
            case ItemEvent.SELECTED:
                System.out.println("CheckBox seleccionado.");
                break;
            case ItemEvent.DESELECTED:
                System.out.println("CheckBox deseleccionado.");
                break;
        }
    }
});
```

Como puedes ver, **ItemEvent** nos permite conocer más detalles de lo que ha ocurrido.

Igual que en el caso de **JButton**, podemos añadir el mismo escuchador a más de un **JCheckBox**, ya que a cada uno de ellos le podemos asignar un *ActionCommand*.

Pero tenemos que hacer algo más de trabajo para conocer quien fue, finalmente, el **JCheckBox** que generó el evento.

```
JCheckBox rojo = new JCheckBox("Rojo");
rojo.setActionCommand("rojo");
ItemListener escuchador;
rojo.addItemListener(escuchador = new ItemListener() {
    @Override
    public void itemStateChanged(ItemEvent e) {
        JCheckBox boton = (JCheckBox)e.getItemSelectable();
        String comando = boton.getActionCommand();
        switch(e.getStateChange()) {
            case ItemEvent.SELECTED:
              System.out.println("Seleccionado el botón " + comando);
              break;
            case ItemEvent.DESELECTED:
```

```
                System.out.println("Deseleccionado el botón " +
comando);
                break;
        }     }     });
JCheckBox amarillo = new JCheckBox("Amarillo");
amarillo.setActionCommand("amarillo");
amarillo.addItemListener(escuchador);
```

Botones con estado y excluyentes

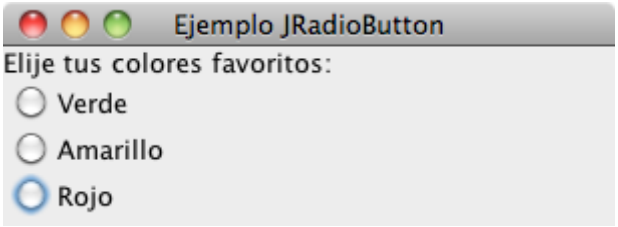
Si lo que necesitas son botones con estado y con comportamiento excluyente, **JRadioButton** y **ButtonGroup** son lo que necesitas.

Ten en cuenta que es una conveción usar **JCheckBox** cuando no necesitamos comportamiento excluyente, y **JRadioButton** cuando sí lo necesitamos.

Por qué los **radio button** se llama así.

Fuente: https://p.twimg.com/Asg7fOeCAAE-4Oo.jpg

Si en el código de ejemplo de **JCheckBox** sustituimos esa por **JRadioButton** el resultado visual es:



Y su comportamiento es exactamente el mismo que antes.

Pero, la diferencia es que podemos *agrupar* los **JRadioButton** para que tengan un comportamiento excluyente.

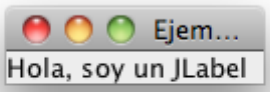
```
ButtonGroup grupo = new ButtonGroup();
grupo.add(verde);
grupo.add(amarillo);
grupo.add(rojo);
```

Un **ButtonGroup** es una agrupación lógica, no tiene ningún resultado visual sobre los **JRadioButton**.

Lo que hemos conseguido, ahora, es que cada vez que un botón se selecciona, el seleccionado anterior, si existe, se deselecciona.

Texto que no se puede modificar

Uno de los componentes más *sencillos* en Swing es **JLabel**, con el podemos visualizar texto que el usuario no puede modificar.

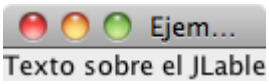


El código para generar la ventana anterior:

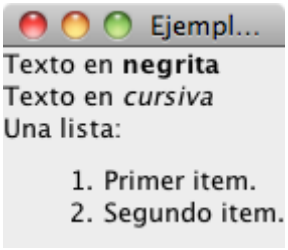
```
JFrame ventana = new JFrame("Ejemplo de JLabel");
ventana.getContentPane().add(new JLabel("Hola, soy un JLabel"));
```

El usuario no puede modificar el texto que aparece sobre el **JLabel**. Aunque sí lo podemos modificar desde nuestro código:

```
JLabel etiqueta = new JLabel();
etiqueta.setText("Texto sobre el JLable");
ventana.getContentPane().add(etiqueta);
```



Los **JLabel** permiten, incluso, usar etiquetas HTML, que sabe como formatear correctamente:




```
String html = "<html>" +
              "Texto en <b>negrita</b><br/>" +
              "Texto en <i>cursiva</i><br/>" +
              "Una lista:" +
              "<ol><li>Primer item.</li>" +
              "<li>Segundo item.</li>" +
              "</ol>" +
              "</html>";
JFrame ventana = new JFrame("Ejemplo de JLabel");
JLabel etiqueta = new JLabel(html);
```

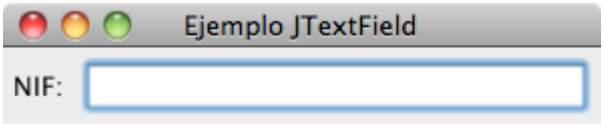
Además, como parte del contenido del **JLabel** se pueden mostrar iconos.

También tenemos opciones para alinear el texto en horizontal y vertical, etcétera.

Puedes encontrar información detallada aquí

Mostrar texto en una única línea

Si lo que necesitas es un componente que permita al usuario introducir una línea de texto, para leer su NIF por ejemplo, puedes utilizar **JTextField**.




```
JFrame ventana = new JFrame("Ejemplo JTextField");
Container contenedor = ventana.getContentPane();
JTextField nif = new JTextField(20);
JLabel nifLabel = new JLabel("NIF: ");
contenedor.setLayout(new FlowLayout());
contenedor.add(nifLabel);
contenedor.add(nif);
ventana.pack();
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
ventana.setVisible(true);
```

Detalle de código:

- Cuando creas el **JTextField** puedes indicar el número de caracteres que contendrá.
- Hemos cambiado el *gestor de aspecto* que por defecto tiene **JFrame** a un **FlowLayout** con:

```
contenedor.setLayout(new FlowLayout());
```

- También hemos añadido un **JLabel**.

Para leer el texto actual de un **JTextField** utilizamos el método **String getText()**.

Para modificar el texto de un **JTextField** utilizamos el método **void setText(String texto)**.

Podemos incluso, decidir si el texto se puede editar o no con **setEditable(boolean sino)**

Además, puedes cambiar la fuente de las letras, etcétera.

Un **JTextField** genera eventos de tipo **ActionListener** cada vez que se pulsa la tecla *Enter* y tiene el foco.

Ya conocemos, de las clases que implementan botones, cómo escuchar este tipo de eventos.

Otro evento que quizás te pueda interesar es **CaretEvent**, que se genera cada vez que desplazamos el cursor sobre el texto que aparece en el **JTextField**.

De nuevo, puedes prever que la **interface** que debes implementar es **CaretListener**, y habrás acertado.

Esta **interface** define un único método **caretUpdate(CaretEvent e)**.

```
nif.addCaretListener(new CaretListener() {
    @Override
    public void caretUpdate(CaretEvent e) {
        System.out.println(e.getDot());
        System.out.println(e.getMark());
    }
});
```

El método **getDot()** nos indica la posición actual del cursor si no hay texto seleccionado, o la posición de un extremo del texto que tengamos seleccionado.

El método **getMark()** nos indica la posición del otro extremo del texto que tengamos seleccionado. Si no hay texto seleccionado, las dos posiciones coinciden.

Mostrar texto en párrafos

Si lo que necesitas es mostrar párrafos de texto, la clase que lo permite es **JTextArea**.

A **JTextArea** le puedes añadir los mismos escuchadores que a **JTextField**.

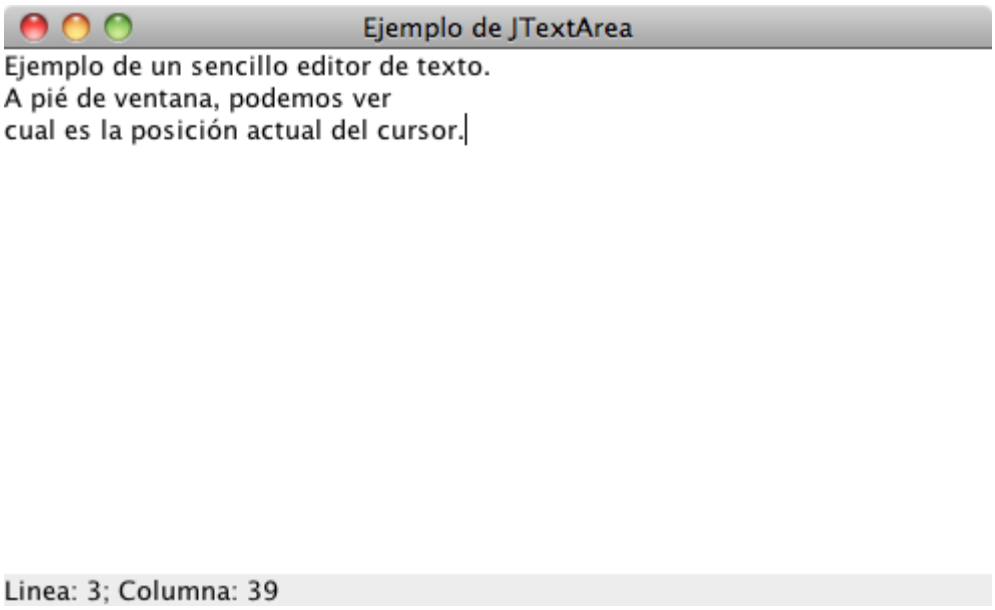
Veamos un ejemplo algo más elaborado con el escuchador de eventos **CaretEvent** y los métodos que nos proporciona la clase **JTextArea**:

```
texto.addCaretListener(new CaretListener() {
```

```
    @Override
    public void caretUpdate(CaretEvent e) {
        int posicion = e.getDot();
        try {
            int linea = texto.getLineOfOffset(posicion);
            int columna = posicion - texto.getLineStartOffset(linea);
            String infoTexto = "Linea: " + (linea+1) +
                               "; Columna: " + (columna+1);
            info.setText(infoTexto);
        } catch (BadLocationException e1) {
            e1.printStackTrace();
        }
    }
});
```

El método **getLineOfOffset(posicion)** del **JTextArea** devuelve la línea a partir de una posición dentro del texto.

El método **getLineStartOffset(linea)** devuelve cual es la posición, dentro del texto del primer carácter de esta línea.



Barras de desplazamiento

Como habrás observado, por defecto un **JTextArea** no tiene barras de desplazamiento -scroll-

Las barras de desplazamiento se pueden añadir a ciertos componentes, como por ejemplo **JTextArea**, o como veremos más adelante, a **JList**.

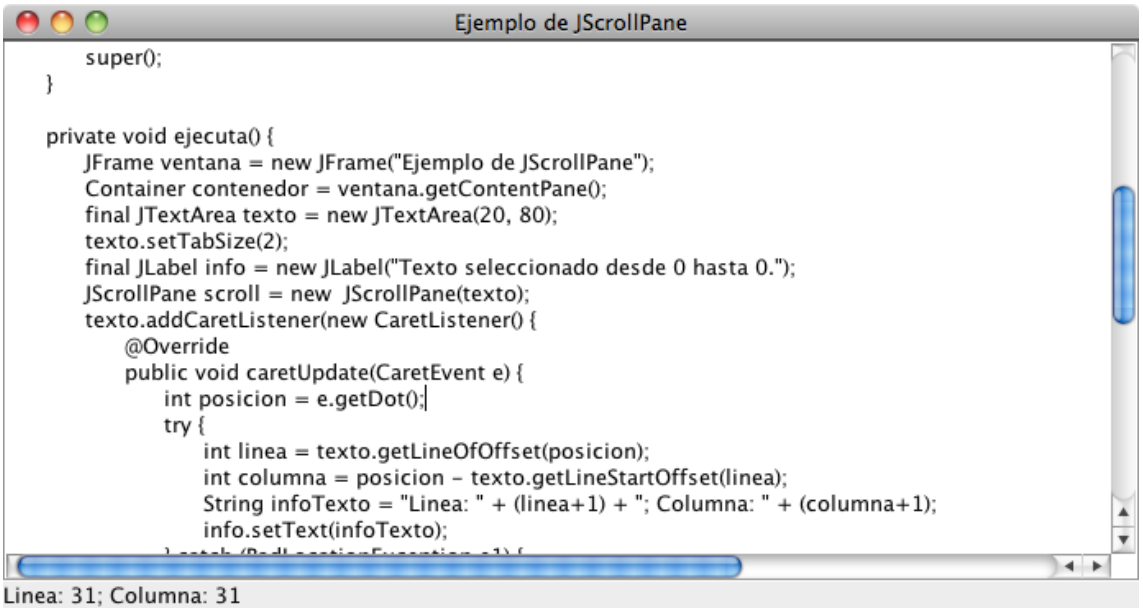
Las barras de desplazamiento implementan el patrón de diseño *Decorador*, son componentes que *decoran* a otros componentes.

Para añadir barras de desplazamiento horizontales y verticales utilizamos **JScrollPane**.

El detalle, en este caso es que, lo que finalmente añadimos a la ventana principal es el **JScrollPane** no el componente que ha decorado.

```
JScrollPane scroll = new  JScrollPane(texto);
...
contenedor.add(scroll);
```

El resultado final es:



JScrollPane nos permite definir la *política* de aparición de las barras de desplazamiento:

- **setHorizontalScrollBarPolicy(int politica)**.
- **setVerticalScrollBarPolicy(int politica)**.

Donde la *política* puede ser:

- HORIZONTAL_SCROLLBAR_AS_NEEDED
- HORIZONTAL_SCROLLBAR_ALWAYS
- HORIZONTAL_SCROLLBAR_NEVER

Si queremos que las barras de desplazamiento se vean sólo cuando es necesario, siempre, o nunca.

Listas

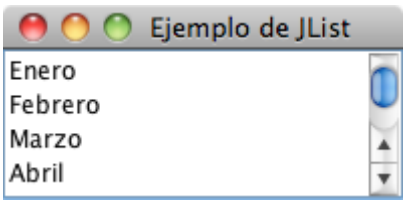
Si necesitas visualizar una lista de elementos, puedes utilizar **JList**.

El código para generar la imagen anterior:

```
String[] datos = {"Enero", "Febrero", "Marzo", "Abril",
                  "Mayo", "Junio", "Julio", "Agosto",
                  "Septiembre", "Octubre", "Noviembre",
                  "Diciembre"};
JFrame ventana = new JFrame("Ejemplo de JList");
JList meses = new JList(datos);
ventana.getContentPane().add(meses);
ventana.pack();
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
ventana.setVisible(true);
```

Como ves, los elementos de la lista los puedes tener en un array.

Al igual que en el caso de **JTextArea**, por defecto la clase **JList** no incluye barras de desplazamiento. Si las necesitamos, las incluimos tal y como hicimos en el caso de **JTextArea**:



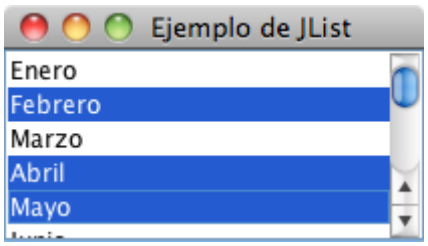
Y puedes asignar el número de elementos visibles con **setVisibleRowCount(int)**

El código para generar la imagen anterior:

```
String[] datos = {"Enero", "Febrero", "Marzo", "Abril",
                  "Mayo", "Junio", "Julio", "Agosto",
                  "Septiembre", "Octubre", "Noviembre",
                  "Diciembre"};
JFrame ventana = new JFrame("Ejemplo de JList");
JList meses = new JList(datos);
JScrollPane scroll = new JScrollPane(meses);
meses.setVisibleRowCount(4);
ventana.getContentPane().add(scroll);
ventana.pack();
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```
ventana.setVisible(true);
```

Por defecto, el **JList** se crea con selección de múltiples intervalos.



Para cambiar el modo de selección puedes utilizar **setSelectionMode(int)** donde los modos de selección son constantes de **interface ListSelectionModel**:

- SINGLE_SELECTION
- SINGLE_INTERVAL_SELECTION
- MULTIPLE_INTERVAL_SELECTION

Si buscamos qué eventos genera **JList** cuando un usuario selecciona alguno de los elementos de la lista, nos encontraremos que el evento es **ListSelectionEvent** que podremos escuchar implementando la **interface ListSelectionListener**.

Esta **interface** sólo declara un método **valueChanged(ListSelectionEvent e)**

ListSelectionEvent nos puede informar, entre otras cosas, si se ha acabado la selección; y en el caso de lista de selección múltiple: el índice del primer elemento seleccionado, y el índice del último elemento seleccionado:

- **getValueIsAdjusting();**
- **getFirstIndex();**
- **getLastIndex();**

Pero si lo que te interesa es consultar, en cualquier momento, por el estado de selección de los elementos, puedes utilizar: **getSelectedValuesList()** método de la clase **JList** que devuelve una lista con todos los elementos seleccionados.

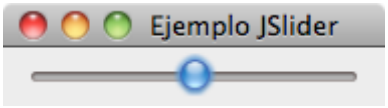
Por su parte **getSelectedIndices()** devuelve un array de enteros con los índices de los elementos seleccionados.

Para recuperar un elemento de la lista a partir de su índice

JList.getModel().getElementAt(int indice).

Sliders

Los **JSlider** nos sirven para cambiar un valor de modo visible.




```
JFrame ventana = new JFrame("Ejemplo JSlider");
Container contenedor = ventana.getContentPane();
JSlider slider = new JSlider(0, 100);
contenedor.add(slider);
ventana.pack();
ventana.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
ventana.setVisible(true);
```

Cada vez que manipulamos un **JSlider** se genera un **ChangeEvent** que podemos escuchar con la **interface ChangeListener** que declara un único método **stateChanged(ChangeEvent e)**

```
slider.addChangeListener(new ChangeListener() {
    @Override
    public void stateChanged(ChangeEvent e) {
        JSlider slider = (JSlider)e.getSource();
            if(slider.getValueIsAdjusting())
                valorActual.setText("Valor actual: " +
                    slider.getValue());
            else valorFijado.setText("Valor fijado: " +
                    slider.getValue());
    }   });
```

Como has visto en el ejemplo, se pueden modificar muchos detalles visuales de un **JSlider**:

```
JSlider slider = new JSlider(0, 100);
slider.setMajorTickSpacing(20);
slider.setMinorTickSpacing(5);
slider.setPaintTicks(true);
slider.setPaintLabels(true);
```

- **setMajorTickSpacing(int)**: intervalo entre *ticks* primarios.
- **setMinorTickSpacing(int)**: intervalo entre *ticks* secundarios
- **setPaintTicks(boolean)**: si se visualizan los *ticks* o no.
- **setPaintLabels(boolean)**: si se visualizan las etiquetas de los *ticks* o no.

Resumen

Hemos visto algunos de los componenetes de una interfaz gráfica de usuario más comunes: Botone, componentes de texto, barras de desplazamietno, listas y slider.

El modelo de programación para la detección de los eventos de usuario está basado en el patrón de diseño *Observer* y el método *push* de notificación.

Un mismo componente, en general, genera más de un tipo de evento cada vez que el usuario interacciona sobre él.

La técnica para programar respuestás a los eventos que generan es siempre la misma: definir una clase que implemente el **interface** capaz de escuchar los eventos en los que estamos interesados.