Introducción a interfaces gráficos de usuario (GUI) y programación dirigida por eventos

Tecnología de la Programación

Curso 2019-2020

Jesús Correas – jcorreas@ucm.es

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

(Basado en material creado por Yolanda García y los apuntes de la asignatura de Marco Antonio Gómez y Jorge Gómez)

Interfaces gráficos de usuario

- Hasta ahora los programas que hemos hecho usan la consola para interactuar con el usuario.
  - En "modo texto": basada en lectura y escritura de caracteres a través del teclado y una pantalla no gráfica (o ventana de texto).
- Esto nos ha permitido centrarnos en los conceptos de programación.
- Sin embargo, Java incluye librerías de clases para hacer aplicaciones con **Interfaz Gráfico de Usuario (GUI)**
  - Basada en ventanas para mostrar información de la aplicación e interactuar con ella.
  - Con otros dispositivos de entrada: por ejemplo con el **ratón**.

Interfaces gráficos de usuario

- En el API de Java existen varios conjuntos de librerías de clases para programar aplicaciones con interfaz gráfico:
  - **AWT:** Es el primer API para GUI de Java. Se basa en los componentes visuales nativos de cada sistema operativo donde se ejecute. No es muy operativo, pues utiliza solamente componentes comunes a todos los sistemas operativos.
  - **Swing:** Es independiente de la plataforma y no utiliza los componentes visuales nativos del sistema operativo.
- Swing utiliza algunas clases de AWT, como la gestión de eventos.
- Veremos la programación GUI con Swing.

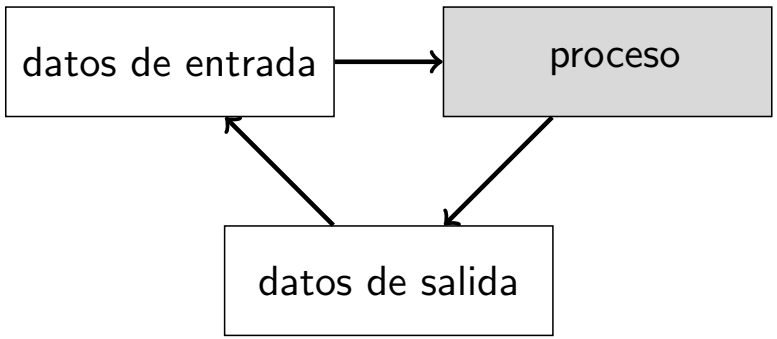
Interfaces gráficos de usuario

- ¿Qué ofrece una librería de interfaz gráfico de usuario?
  - Herramientas para crear ventanas gráficas y cuadros de diálogo.
  - Un conjunto de componentes visuales: botones, entrada de datos, *checkbox*, listas desplegables, etc.
  - Un sistema para la gestión de eventos.
  - Como característica adicional, se necesita (y se ofrece) la capacidad de ejecución de algunas tareas en varios hilos de ejecución.
- El diseño de los programas suele realizarse utilizando el paradigma de **programación dirigida por eventos.**

Interfaces gráficos de usuario

- Hasta ahora los programas que hemos hecho usan la consola para interactuar con el usuario.
- Siempre siguen un esquema similar a los siguientes:

proceso simple




```
Scanner sc = new Scanner(new File(
    "fichero1"));...
 while (...) {
    String s = sc.next();
    if (s.equals("...")) {
      //caso 1
    } else if (s.equals("...")) {
      //caso 2
    } else ...}
```

- La **interacción con el usuario** está **controlada por el programa.**

Interfaces gráficos de usuario

- Este modelo de interacción es muy rígido para el usuario.
- Los programas con interfaces gráficas de usuario (GUI) son muy difíciles de programar con este modelo de interacción.
- Las aplicaciones gráficas se centran en la **usabilidad**: **el usuario es el que decide lo que debe realizar el programa** en cada momento.
- Se utiliza el concepto de **inversión del control:**
  - El programa ofrece al usuario una serie de *controles* o *componentes visuales* (botones, menús, barras de herramientas, etc.).
  - El usuario puede utilizar estos controles en cualquier orden.
  - Se debe intentar que el usuario pueda interactuar en todo momento con todos los controles (en la medida de lo posible).
- Como resultado, **no existe un flujo de ejecución determinado.** El orden de las funcionalidades depende de las decisiones del usuario.
- La programación de este tipo de aplicaciones es más compleja.

Ejemplo 1

- Un ejemplo mínimo de programa con interfaz gráfico de usuario es el siguiente:

```
import javax.swing.JFrame;

public class Ejemplo1 {
  public static void main(String []args) {
    JFrame ventana = new JFrame("Mi primera ventana - Ejemplo 0");
    ventana.setSize(620, 200);
    ventana.setVisible(true);
  }
}
```

- Este programa no funciona del todo bien: no termina cuando se cierra la ventana.
- Además, la ventana se debe hacer visible en el hilo de trabajo de Swing (lo veremos más adelante).

Ejemplo 2

- En la versión anterior se crea un objeto de tipo `JFrame` y se invocan métodos sobre él.
- Otra forma habitual de programar aplicaciones de ventanas es creando clases que hereden de `JFrame` (u otros componentes):

```
import javax.swing.*;
public class Ejemplo2 extends JFrame {
  public Ejemplo2 () {
       super("Mi primera ventana - Ejemplo 2");
       initGUI();}
  private void initGUI(){
       this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       this.setSize(620, 200);
       this.setVisible(true);//aquí arranca el hilo de Swing
  }
  public static void main(String []args) {
       SwingUtilities.invokeLater(new Runnable() {
        public void run() {
             new Ejemplo2();//creación ventana en el hilo de Swing
        }});
  }
}
```

Ejemplo 3

- En la siguiente versión del programa vamos a crear un botón y añadirlo a la ventana:

```
public class Ejemplo3 extends JFrame {
   public Ejemplo3() {
      super("Mi primera ventana - Ejemplo3");
      this.setSize(320, 200);
      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
      JButton boton = new JButton("Pulsa para saludar");
      boton.addActionListener(new MiActionListener());
      this.getContentPane().add(boton);
      v.setVisible(true);}

   public class MiActionListener implements ActionListener {
      public void actionPerformed(ActionEvent event) {
            System.out.println("Hola Mundo!");}}

   public static void main(String []args) {
      SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                  new Ejemplo3();}
      });
   }
}
```

Programación dirigida por eventos

- En las aplicaciones GUI se utiliza un modelo de programación **dirigida por eventos**.
- Un **evento** es **cualquier suceso relevante para la aplicación.**
- La aplicación muestra los controles del interfaz GUI al usuario.
- Los *componentes visuales* de la aplicación son objetos del programa que pueden **generar eventos** como resultado de acciones del usuario.
- Para los eventos que sean relevantes, en el código de la aplicación se crean **objetos** que realizan la funcionalidad que corresponda a cada evento. Estos objetos se denominan **listeners** (manejadores u oyentes).
- Los objetos que generan eventos **no acceden directamente a los objetos manejadores:** Se utiliza un *mecanismo de delegación*.
- Los objetos manejadores **se registran** en el objeto que genera el evento para que éste los avise cuando se produce el evento.

Cómo funciona la gestión de eventos en Java

- Los componentes visuales son instancias de clases (o subclases de éstas) de las librerías de Java.
- Estas clases contienen el código necesario para determinar cuándo el usuario realiza una acción sobre el objeto correspondiente (normalmente, interactuando con el SO).
  - Cuando se pulsa el botón del ratón sobre el área ocupada por un botón de una ventana.
  - También cuando el ratón pasa por encima de un componente visual, cuando se pulsa una tecla, etc.
- Para ello, el objeto del componente visual mantiene **una lista de listeners por cada tipo de evento** que se puede producir.
- El programador debe añadir código en el programa para añadir a esa lista el (o los) objetos que serán listeners del evento correspondiente. En el ejemplo anterior:

```
JButton boton = new JButton("Pulsa para saludar");
boton.addActionListener( /*objeto listener*/ );
```

Cómo funciona la gestión de eventos en Java

- Cuando ocurre una acción, se genera un evento y se **avisa** a los listeners registrados.
- La forma de avisarlos es ejecutando determinado método, según se indica en el interfaz correspondiente.
- **En el ejemplo,** el evento que se debe capturar es de tipo **`ActionEvent`**, y el listener debe implementar el interfaz **`ActionListener`**.
- Este interfaz contiene una declaración de método:

```
public interface ActionListener extends EventListener {
    /**
     * Invoked when an action occurs.
     */
    public void actionPerformed(ActionEvent e);
}
```

- Por tanto, el código que queremos que se ejecute cuando se pulsa el botón debe estar **en una clase que implemente este método**, y en él debe contener el código a ejecutar.

Cómo funciona la gestión de eventos en Java

- En el código del ejemplo anterior, hemos creado una clase anónima para realizar esto en el argumento de la llamada a **addActionListener**:

```
...
boton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent event) {
    System.out.println("Hola Mundo!");
  }});
```

- El uso de clases anónimas es opcional: si el código es más complejo, se puede utilizar una instancia de una clase local, interna, o una clase no anidada:

```
    boton.addActionListener(new AccionSaludo());
    ...
  } }

public class AccionSaludo implements ActionListener {
  public void actionPerformed(ActionEvent event) {
    System.out.println("Hola Mundo!");}
}
```

Tipos de eventos e interfaces manejadores

- Se pueden producir distintos tipos de eventos, que se representan en el programa mediante objetos de clases de los paquetes **`java.awt.event`** y **`javax.swing.event`**. Por ejemplo:

Evento	Interfaz y método manejador
`ActionEvent`	**`ActionListener`** `void actionPerformed (ActionEvent e)`
`ChangeEvent`	**`ChangeListener`** `void stateChanged (ChangeEvent e)`
`ItemEvent`	**`ItemListener`** `void itemStateChanged (ItemEvent e)`
`ListSelectionEvent`	**`ListSelectionListener`** `void valueChanged (ListSelectionEvent e)`

Tipos de eventos e interfaces manejadores

- Cada uno de los tipos de eventos tiene distintos métodos que proporcionan información sobre el evento.
- Por ejemplo, **`ActionEvent:`**
  - Se produce cuando se realiza la *acción estándar* sobre determinados componentes. Por ejemplo, cuando se pulsa con el ratón sobre un botón.
- **`ItemEvent:`**
  - Se produce cuando se selecciona un elemento de una lista desplegable (`JComboBox`) o un `JCheckbox`.
  - El método **`getItem()`** devuelve el ítem seleccionado.
- Existen más eventos, como **`ChangeEvent`**, por ejemplo cuando se pasa el ratón por encima del botón(sin hacer click).
- El evento **`ListSelectionEvent`** se produce cuando hay un cambio en la selección actual de una lista.

Tipos de componentes visuales

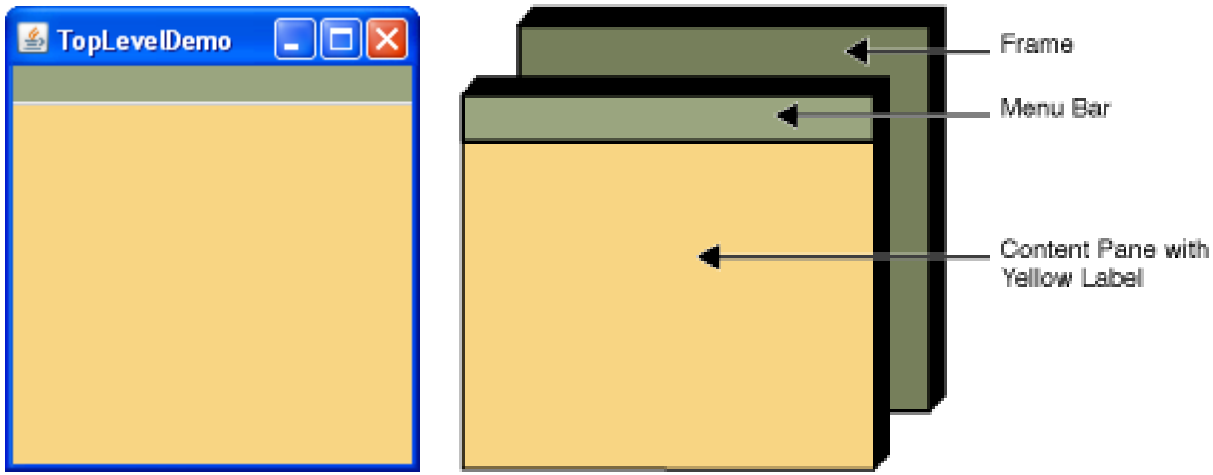
- Cuando se programa una aplicación GUI, los componentes pueden estar unos contenidos en otros (por ejemplo, una ventana contiene botones, campos de texto, etc.).
- Mantienen una **jerarquía** respecto a este aspecto:
  - **Contenedores de primer nivel** (*top-level containers*) son componentes que no pueden estar contenidos en ningún otro componente: **`JFrame`**,**`JDialog`**.
  - **Contenedores intermedios:** agrupan otros componentes y permiten realizar acciones comunes sobre ellos: **`JPanel`**, **`JScrollPane`**.
  - **Componentes:** son los *controles* habituales: **`JLabel`**, **`JButton`**, **`JTextField`**, **`JTextArea`**, etc.
- Además, hay muchas otras clases para tratar diversos detalles de los componentes visuales: **`Graphics`**, **`Color`**, **`Font`**, **`layouts`**, etc.

Contenedores de primer nivel

- Cualquier componente de una aplicación Swing que se muestre en la pantalla debe formar parte de una **jerarquía de componentes** (*containment hierarchy*).
- Una aplicación Swing puede tener varias jerarquías, cada una de ellas **con un contenedor de primer nivel en la raíz.**
- Contenedores de primer nivel:
  - **`JFrame`** es una ventana no contenida en otras ventanas, con barra de título y botones de control de la ventana (cerrar, maximizar, etc.).
  - **`JDialog`** es un tipo de ventana especial: un **cuadro de diálogo**.

Componentes visuales de Swing

- Una ventana (componente de primer nivel) tiene diversos elementos:



- En particular tiene un **contenedor asociado:** el **content pane**. Sobre este contenedor se muestran los demás componentes contenidos en la ventana.

Disposición de los componentes en una ventana

- En el panel de una ventana se pueden añadir componentes. En el ejemplo 3 hemos visto la forma de añadir un botón:

```
public class Ejemplo3 extends JFrame {
  public Ejemplo3() {
    ...
    JButton boton = new JButton("Pulsa para saludar");
    ...
    this.getContentPane().add(boton);
  } }
```

- El botón **ocupa todo el espacio de la ventana.**
- Por defecto los componentes **se redimensionan automáticamente** para ocupar el espacio del contenedor en el que se encuentran.
- Para disponer varios componentes en la ventana (y en general en un contenedor) se utiliza objetos que representan **esquemas de disposición de componentes** denominados **layouts**.
- Hay varios tipos de *layouts*, los más comunes son: **`FlowLayout`**, **`BorderLayout`**, **`GridLayout`** y **BoxLayout**

Disposición de los componentes en una ventana

- Para establecer el *layout* de un contenedor se utiliza el método **`setLayout(LayoutManager mgr)`**. Por ejemplo:

```
public class EjemploFlowLayout extends JFrame {
  public EjemploFlowLayout() {
    ...
    JButton boton = new JButton("Pulsa para saludar");
    ...
    this.getContentPane().setLayout(new FlowLayout());
    this.getContentPane().add(new JLabel("Ventana de saludo"));
    this.getContentPane().add(boton);
    this.getContentPane().add(new JLabel("otra etiqueta"));
  } }
```

- **`FlowLayout`** muestra los componentes de izquierda a derecha y de arriba abajo.
- Ver **`EjemploFlowLayout.java`**.

Disposición de los componentes en una ventana

- Otro *layout* es **`BorderLayout.`**
- Divide la ventana en **cinco regiones**: norte, sur, este, oeste y centro.

- Para añadir un componente a una de las regiones, se debe utilizar el método **`add(Component, int)`**:

```
this.getContentPane().setLayout(new BorderLayout());
JLabel lblEste = new JLabel("region este");
lblEste.setBackground(Color.GREEN);
lblEste.setOpaque(true);
this.getContentPane().add(lblEste,BorderLayout.EAST);
...
this.getContentPane().add(lblOeste,BorderLayout.WEST);
...
```

- Ver **`EjemploBorderLayout.java.`**

Disposición de los componentes en una ventana

- El *layout* **`GridLayout`** dispone los componentes en una cuadrícula, con una serie de filas y de columnas (**`EjemploGridLayout .java`**).

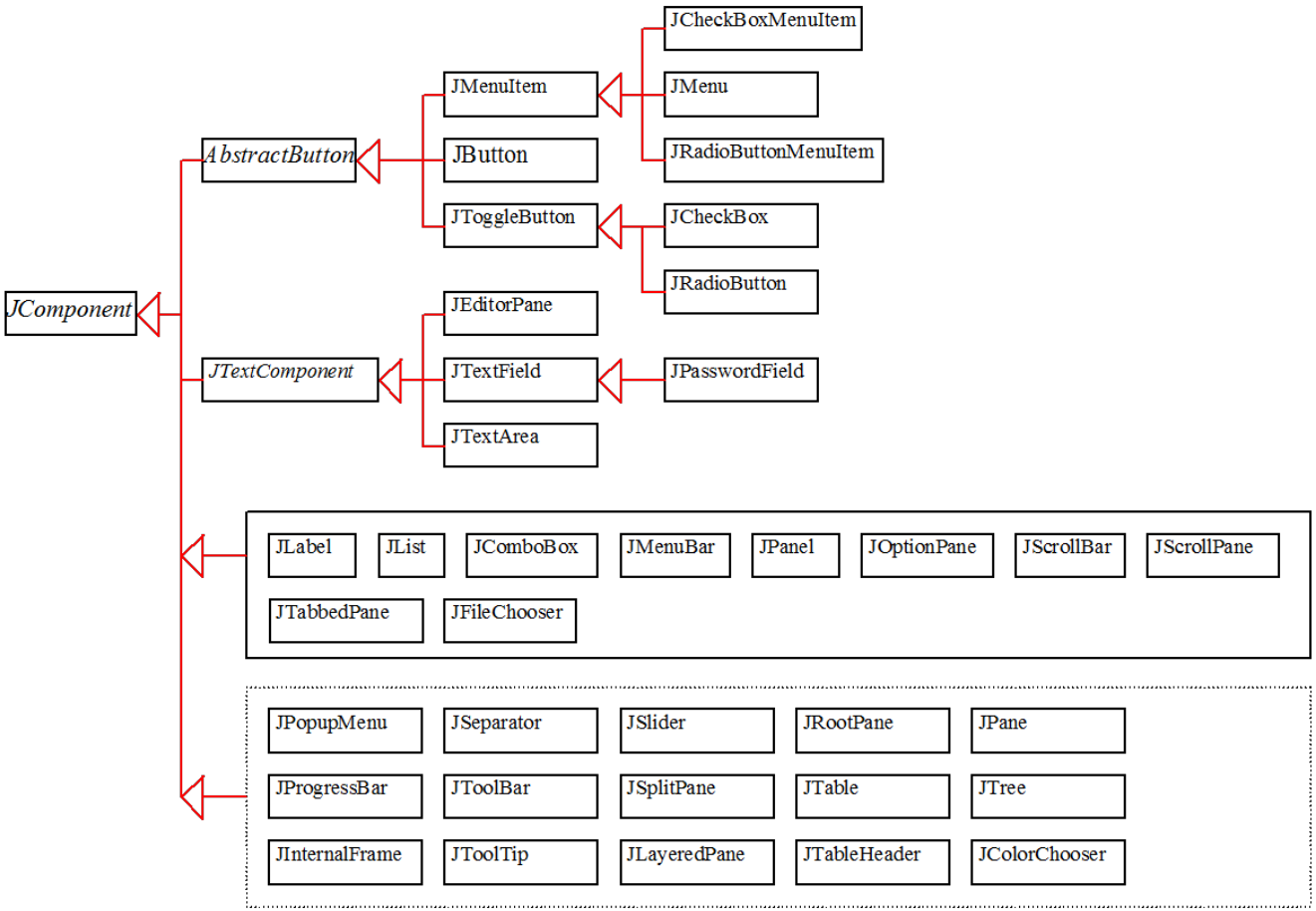
- Los componentes se van añadiendo al contenedor de forma ordenada por filas (o por columnas).
- Se le puede indicar el espacio entre regiones.

```
this.getContentPane().setLayout(new GridLayout(filas,cols,5,5));
for (int i = 0; i < filas; i++) {
  for (int j = 0; j < cols; j++) {
    this.getContentPane().add(
      new JLabel("("+i+","+j+")"));
  }
}
```

Agrupamiento de componentes. Contenedores

- Se pueden utilizar varios *layouts* en una misma ventana, unos dentro de otros.
- Por ejemplo, si dentro de una región de un `BorderLayout` queremos disponer una serie de botones en forma de cuadrícula (con un `GridLayout`) (por ejemplo, un teclado numérico para una calculadora).
- Para ello, dentro de una región debemos añadir un **contenedor** que nos permita establecer otro *layout*. Por ejemplo, un **`JPanel`**.
- Una vez añadido este componente, se le puede establecer un *layout* diferente, y añadir componentes sobre él.
- Ver **`EjemploAgrupComponentes.java`**

Componentes visuales de Swing



- Veremos algunos de ellos.

Algunas subclases de `JComponent`

Además de los componentes que hemos visto hasta ahora (**`JFrame`**, **`JPanel`**, **`JButton`**, **`JLabel`**), hay otros componentes que se utilizan muy frecuentemente:

- **`JTextField`**: muestra un cuadro para que el usuario pueda introducir un texto de una sola línea.
- **`JTextArea`**: muestra un cuadro de texto en el que el usuario puede introducir un texto de varias líneas.
- **`JCheckBox`**: muestra un pequeño cuadro con una etiqueta de texto en el que el usuario puede activar o desactivar una opción.
- **`JRadioButton`**: muestra un pequeño círculo con una etiqueta de texto en el que el usuario puede activar o desactivar una opción. Estos componentes se utilizan en grupo utilizando un objeto de la clase **`ButtonGroup`**, para que las opciones sean excluyentes, de forma que solo una esté seleccionada.

Algunas subclases de `JComponent`

- Listas:
  - **`JList`**: muestra una lista de elementos para que el usuario seleccione uno o varios de ellos.
  - **`JComboBox`**: muestra una lista desplegable. El usuario puede seleccionar un solo elemento, que es el que aparecerá en la lista sin desplegar.
- Barras de menús y menús:
  - **`JMenuBar`**: muestra una barra de menú que aparece en la parte superior de una ventana o contenedor de primer nivel (no se puede utilizar en otros contenedores).
  - **`JMenu`**: es un menú (de una barra de menú o de un menú *pop-up*). Cada elemento del menú es un componente **`JMenuItem`**.
  - **`JPopupMenu`**: es un menú de contexto, que se activan al pulsar el botón derecho del ratón sobre otro componente.

Algunas subclases de `JComponent`

- Cuadros de diálogo: Ventanas especiales con las que se tiene más control sobre las acciones del usuario.
- Heredan de **`JDialog`**.
- Algunos cuadros de diálogo preconfigurados son:
  - **`JOptionPane`**: muestra cuadros de diálogo sencillos con botones por defecto del tipo **Sí/No/Cancelar**, o **Aceptar/Cancelar**.
  - **`JFileChooser`**: muestra un cuadro de diálogo para seleccionar un archivo. Se puede configurar para seleccionar la extensión, directorio, etc.
  - **`JColorChooser`**: muestra un cuadro de diálogo para seleccionar un color de la paleta de colores.
- Un ejemplo más completo con algunos componentes sencillos: **`EjemploConversorMoneda.java`**.