

Componentes Visuales



Curso 2024/25

Repaso Java

Clases abstractas

Clases abstractas

Son clases “molde”, es decir clases de las cuales se puede heredar pero **nunca** se pueden crear objetos de dicha clase.

Aunque no se pueden crear objetos de una clase abstracta, sí **es posible** declarar objetos de ese tipo.

Una clase es abstracta si:

- Contiene un método abstracto (un método que no contiene implementación, sólo su declaración). El método puede ser heredado.
- Si va precedida de la palabra `abstract`. En tal caso, podría ocurrir que ni siquiera tuviera métodos abstractos.

Clases abstractas

```
abstract public class A {  
    protected int x;  
    A(){  
        this.x = 0;  
    }  
    A(int xx){  
        this.x = xx;  
    }  
    public int getX(){  
        return this.x;  
    }  
    abstract public void increment();  
}
```

```
abstract public class A1 extends A {  
    public A1(int xx){  
        super(xx);  
    }  
    // No se implementa el método increment  
    // La clase tiene que ser necesariamente abstracta  
}
```

Clases abstractas

```
public class A2 extends A1 {  
    private int y;  
    public A2(int xx,int yy){  
        super(xx);  
        this.y = yy;  
    }  
    public void increment() {  
        this.y = this.y + this.x;  
    }  
    public int getY() {  
        return this.y;  
    }  
}
```

Clases abstractas

```
public class A2 extends A1 {  
    private int y;  
    public A2(int xx,int yy){  
        super(xx);  
        this.y = yy;  
    }  
    public void increment() {  
        this.y = this.y + this.x;  
    }  
    public int getY() {  
        return this.y;  
    }  
}
```

```
A a = new A(2);  
A a = new A1(2);  
A a = new A2(2,3);  
a.increment();  
System.out.println(a.getX());  
System.out.println(a.getY());
```

Clases abstractas

```
public class A2 extends A1 {  
    private int y;  
    public A2(int xx,int yy){  
        super(xx);  
        this.y = yy;  
    }  
    public void increment() {  
        this.y = this.y + this.x;  
    }  
    public int getY() {  
        return this.y;  
    }  
}
```

```
A a = new A(2);           // ¿Correcto?  
A a = new A1(2);         // ¿Correcto?  
A a = new A2(2,3);       // ¿Correcto?  
a.increment();           // ¿Correcto?  
System.out.println(a.getX()); // ¿Correcto?  
System.out.println(a.getY()); // ¿Correcto?
```

Repaso Java

Interfaces

Interfaces y clases abstractas: diferencias

- Todos los métodos de una interfaz se declaran **implícitamente** abstractos y públicos.
- Una interfaz no puede implementar **ningún** método y una clase abstracta sí.
- Una interfaz **no tiene** atributos **salvo** que sean estáticos o constantes. Además, implícitamente siempre son públicos.
- Una clase puede implementar **varias** interfaces pero sólo puede extender a **una** clase.
- Una clase abstracta puede pertenecer a una jerarquía de clases mientras que una interfaz no. Por lo tanto, clases no relacionadas pueden implementar la misma interfaz.
- Una interfaz sólo puede **extender** a otras interfaces. Una clase puede **extender** otras clases e **implementar** distintas interfaces.

Sintaxis

La sintaxis de una interfaz es la siguiente:

Deben declararse en un archivo con el mismo nombre y extensión `.java`.

Sintaxis

La sintaxis de una interfaz es la siguiente:

```
public interface nombre_interfaz [extends I1, ..., In] {  
    [public abstract] tipo_retorno nombre_método (argumentos);  
}
```

Deben declararse en un archivo con el mismo nombre y extensión .java.

Sintaxis

```
public interface FiguraGeometrica {  
    final float PI = 3.1416;  
    float area();  
    void drawFiguraGeometrica();  
}
```

Sintaxis

```
public interface FiguraGeometrica {  
    final float PI = 3.1416;  
    float area();  
    void drawFiguraGeometrica();  
}
```

```
public class Cuadrado implements FiguraGeometrica {  
    private float lado;  
    public Cuadrado(float lado) {  
        this.lado = lado;  
    }  
    public float area() {  
        return lado * lado;  
    }  
    public void drawFiguraGeometrica() {  
        System.out.println("Longitud lado cuadrado: " + this.lado);  
    }  
}
```

```
public class Circulo implements FiguraGeometrica {  
    private float radio;  
    public Circulo(float radio) {  
        this.radio = radio;  
    }  
    public float area() {  
        return PI * radio * radio;  
    }  
    public void drawFiguraGeometrica() {  
        System.out.println("Radio del círculo: " + this.radio);  
    }  
}
```

Implementación

Una clase **puede** implementar una o varias interfaces.

Implementar una interfaz significa **implementar** métodos de la interfaz. Si la clase no implementa algún método de la interfaz, entonces la clase es necesariamente abstracta.

La clase, además, puede tener métodos propios. Si no implementa algún método, entonces es abstracta.

Implementación

Para poder utilizar una constante declarada en una interfaz, o un atributo estático, es necesario **anteponer** el nombre de la interfaz a la constante.

Tarea: crea una interfaz y una clase que la implemente. Declara constantes y/o atributos estáticos en ella y observa cómo funcionan.

La jerarquía entre interfaces permite la herencia simple y múltiple.

- Una clase puede implementar varias interfaces.
- Una interfaz puede “extender” a otras interfaces, pero no puede implementarlas.
- Una interfaz no puede tener métodos estáticos (un método estático no puede ser abstracto).

Implementación

Una clase puede, **simultáneamente**, heredar de otra clase e implementar **una o varias** interfaces.

Una interfaz es un tipo, por lo tanto, el tipo se puede utilizar en cualquier lugar donde se pueda utilizar el identificador de un tipo o de una clase.

Implementación

Una clase puede, **simultáneamente**, heredar de otra clase e implementar **una o varias** interfaces.

```
public class Clase extends SuperClase implements I1, ... In {  
    // Implementación de la clase  
}
```

Una interfaz es un tipo, por lo tanto, el tipo se puede utilizar en cualquier lugar donde se pueda utilizar el identificador de un tipo o de una clase.

Java

Ejercicio clases abstractas

Enunciado ejercicio clase abstracta

Dibuja el diagrama UML antes de codificar. El ejercicio consiste en crear una clase abstracta llamada **ProductoElectronico** con los atributos: **nombre** y **precio**, y el método abstracto **mostrarInformacion()**.

Crea dos clases concretas que extiendan de **ProductoElectronico**:

- **Telefono**: debe tener un atributo adicional llamado **modelo** y debe implementar el método abstracto para que muestre el **nombre**, **modelo** y **precio** del teléfono.
- **Laptop**: debe tener un atributo adicional llamado **marca** y debe implementar el método abstracto para que muestre el **nombre**, **marca** y **precio** del laptop.

En la clase principal **Main**, crea instancias de **Telefono** y **Laptop**, llamando al método **mostrarInformacion()** para cada instancia. Analiza los resultados obtenidos.

Java

Ejercicio interfaces

Enunciado ejercicio interfaz

Dibuja el diagrama UML antes de codificar. El ejercicio consiste en crear una interfaz llamada **Prestable** con los siguientes métodos:

- `prestar()`
- `devolver()`
- `estaPrestado()`

Crea dos clases concretas que implementen dicha interfaz:

- **Libro**: debe tener los atributos `autor` y `isbn`. Además, debe implementar los métodos de la interfaz.
- **Revista**: debe tener los atributos `numero` y `editorial`. Además, debe implementar los métodos de la interfaz.

En la clase principal **Main**, crea instancias de **Libro** y **Revista**. Juega un poco con el código y realiza préstamos y devoluciones (verificando el estado de los préstamos) haciendo uso de los métodos que has implementado. Para ello, crea diferentes instancias de **Libro** y **Revista**.

Java

Clases internas

Definición

Una clase interna es una clase que está definida dentro de otra clase. A veces, también se les conoce como clases anidadas. Las clases internas se utilizan para agrupar de manera lógica las clases que están sólo para el uso de la clase externa (veremos ejemplos a continuación).

Una clase interna puede acceder a los elementos de una clase externa, incluso si estos son privados.

Tipos de clases internas

- ❑ **Clase interna:** es una clase anidada que se declara como static. No puede acceder a los elementos de la clase externa a no ser que estos sean static.
- ❑ **Clase interna estática:** lo mismo que lo anterior pero para clases que no son static.
- ❑ **Clase local:** son clases que se definen dentro de un método, por lo que su ámbito está restringido a dicho método.
- ❑ **Clase anónima:** clases que no tienen nombre. Las veremos en detalle en las próximas diapositivas.

Ejemplo clase interna estática

```
class Externa {  
    static String mensaje = "Hola desde la clase externa!";  
  
    static class InternaEstatica {  
        void mostrarMensaje() {  
            System.out.println(mensaje); // Puede acceder a elementos estáticos  
        }  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Externa.InternaEstatica interna = new Externa.InternaEstatica();  
        interna.mostrarMensaje();  
    }  
}
```

Ejemplo clase interna ~~private~~

```
class Externa {
    private String mensaje = "Hola desde la clase externa!";

    class Interna {
        void mostrarMensaje() {
            System.out.println(mensaje); // Puede acceder a elementos privados
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Externa externa = new Externa();
        Externa.Interna interna = externa.new Interna();
        interna.mostrarMensaje();
    }
}
```

Ejemplo clase local

```
class Externa {  
    void metodoExterno() {  
        class Local {  
            void mostrarMensaje() {  
                System.out.println("Hola desde la clase local!");  
            }  
        }  
  
        Local local = new Local();  
        local.mostrarMensaje();  
    }  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Externa externa = new Externa();  
        externa.metodoExterno();  
    }  
}
```

Ejemplo clase anónima

```
interface Saludo {  
    void saludar();  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Saludo saludo = new Saludo() {  
            public void saludar() {  
                System.out.println("Hola desde la clase anónima!");  
            }  
        };  
        saludo.saludar();  
    }  
}
```

Java

Clases anónimas

Clases anónimas

Las clases anónimas son un subconjunto de clases internas. Es decir, son un tipo de clases internas en concreto.

- Son clases **sin nombre** que se definen e instancian **a la vez**.
- Únicamente se puede crear **una** instancia de la clase.
- Usan la sintaxis especial del operador new.
- Este tipo de clases son usadas para crear objetos “al vuelo” para devolver valores, pasar argumentos o inicializar atributos.

Clases anónimas

- **Siempre** extenderán de otra clase o implementarán una interfaz, de manera que la definición sobrescribe o implementa esos métodos que serán los que puedan llamarse desde fuera.
- No se puede especificar accesibilidad porque son locales al bloque.
- Sólo pueden ser `static` en bloques estáticos.

Clases anónimas

- Guardan una **referencia oculta** a la instancia de la clase externa.
- Tienen acceso a los parámetros y variables locales `final` del bloque.
- No se pueden usar **fuera del bloque** que las define, por lo que no tiene sentido darles nombre. De hecho, no se puede usar algo como: `Clase.ClaseLocal`.
- Las clases anónimas **no pueden** tener constructores ya que, al fin y al cabo, no tienen nombre.

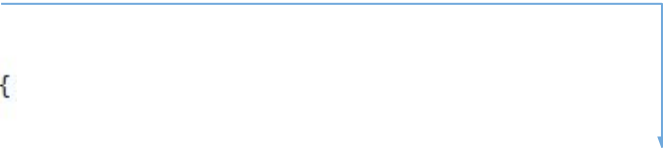
Ejemplo sin usar clases anónimas

```
/**
 * Clase Main
 * @author J. Carrero
 */
public class Main {
    /**
     * Punto de entrada del programa
     * @param args
     */
    public static void main(String[] args) {
        Circulo myCirculo = new Circulo();
        myCirculo.dibujarForma("Circulo");
    }
}
```

```
/**
 * Clase Circulo
 * @author J. Carrero
 */
public class Circulo implements IFormas {
    @Override
    public void dibujarForma(String _figura) {
        System.out.println("Metodo dibujarForma() del " + _figura + " ejecutado.");
    }
    @Override
    public void mostrarPropiedades(String _figura) {
        System.out.println("Metodo mostrarPropiedades() del " + _figura + " ejecutado.");
    }
}
```

Ejemplo sin usar clases anónimas

```
/**
 * Clase Main
 * @author J. Carrero
 */
public class Main {
    /**
     * Punto de entrada del programa
     * @param args
     */
    public static void main(String[] args) {
        Circulo myCirculo = new Circulo();
        myCirculo.dibujarForma("Circulo");
    }
}
```

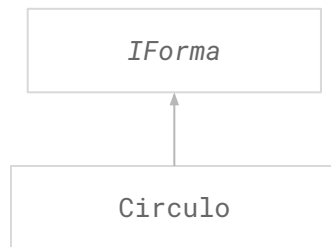


```
/**
 * Clase Circulo
 * @author J. Carrero
 */
public class Circulo implements IFormas {
    @Override
    public void dibujarForma(String _figura) {
        System.out.println("Metodo dibujarForma() del " + _figura + " ejecutado.");
    }
    @Override
    public void mostrarPropiedades(String _figura) {
        System.out.println("Metodo mostrarPropiedades() del " + _figura + " ejecutado.");
    }
}
```

Ejemplo sin usar clases anónimas

```
/**
 * Clase Main
 * @author J. Carrero
 */
public class Main {
    /**
     * Punto de entrada del programa
     * @param args
     */
    public static void main(String[] args) {
        Circulo myCirculo = new Circulo();
        myCirculo.dibujarForma("Circulo");
    }
}
```

```
/**
 * Clase Circulo
 * @author J. Carrero
 */
public class Circulo implements IFormas {
    @Override
    public void dibujarForma(String _figura) {
        System.out.println("Metodo dibujarForma() del " + _figura + " ejecutado.");
    }
    @Override
    public void mostrarPropiedades(String _figura) {
        System.out.println("Metodo mostrarPropiedades() del " + _figura + " ejecutado.");
    }
}
```

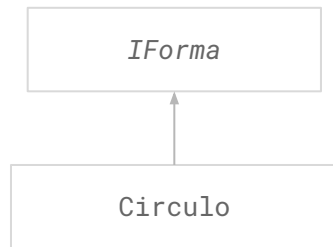


Ejemplo sin usar clases anónimas

```
/**
 * Clase Main
 * @author J. Carrero
 */
public class Main {
    /**
     * Punto de entrada del programa
     * @param args
     */
    public static void main(String[] args) {
        Circulo myCirculo = new Circulo();
        myCirculo.dibujarForma("Circulo");
    }
}
```

¿Y si quiero crear otra forma?

```
/**
 * Clase Circulo
 * @author J. Carrero
 */
public class Circulo implements IFormas {
    @Override
    public void dibujarForma(String _figura) {
        System.out.println("Metodo dibujarForma() del " + _figura + " ejecutado.");
    }
    @Override
    public void mostrarPropiedades(String _figura) {
        System.out.println("Metodo mostrarPropiedades() del " + _figura + " ejecutado.");
    }
}
```



Ejemplo usando clases anónimas

```
/**
 * Clase Main
 * @author J. Carrero
 */
public class Main {
    /**
     * Punto de entrada del programa
     * @param args
     */
    public static void main(String[] args) {
        /**
         * Clase anónima
         */
        IFormas figura = new IFormas() {
            @Override
            public void dibujarForma(String _figura) {
                System.out.println("Metodo dibujarForma() del " + _figura + " ejecutado.");
            }
            @Override
            public void mostrarPropiedades(String _figura) {
                System.out.println("Metodo mostrarPropiedades() del " + _figura + " ejecutado.");
            }
        };
        figura.dibujarForma("Cuadrado");
        figura.dibujarForma("Rectangulo");
        figura.dibujarForma("Triangulo");
    }
}
```

Ejemplo usando clases anónimas

```
/**
 * Clase Main
 * @author J. Carrero
 */
public class Main {
    /**
     * Punto de entrada del programa
     * @param args
     */
    public static void main(String[] args) {
        /**
         * Clase anónima
         */
        IFormas figura = new IFormas() {
            @Override
            public void dibujarForma(String _figura) {
                System.out.println("Metodo dibujarForma() del " + _figura + " ejecutado.");
            }
            @Override
            public void mostrarPropiedades(String _figura) {
                System.out.println("Metodo mostrarPropiedades() del " + _figura + " ejecutado.");
            }
        };
        figura.dibujarForma("Cuadrado");
        figura.dibujarForma("Rectangulo");
        figura.dibujarForma("Triangulo");
    }
}
```

Debe implementar los métodos de la interfaz

La clase anónima no tiene nombre

Accesibilidad limitada al bloque

Java

Interfaces gráficas

Hasta ahora...

Hasta ahora hemos usado programas que utilizan la **consola** para interactuar con el usuario.

Esa forma nos ha permitido **centrarnos** en todo lo que tiene que ver con la programación.

Así estamos dejando que la aplicación se ejecute por sí misma y hacemos que el usuario **solo** intervenga cuando la aplicación lo demanda.

Tipos de programas

Diferenciamos básicamente dos tipos de programas:

Tipos de programas

Diferenciamos básicamente dos tipos de programas:

Modo consola

Interacción con el usuario

Interfaz basado en texto

Aplicaciones con interfaz gráfica

Ventanas gráficas para
entrada/salida de datos

Iconos

Interacción directa

Interfaces gráficas de usuario (GUI)

Las GUI ofrecen, entre otras cosas:

- ❑ Ventanas
- ❑ Cuadros de diálogo
- ❑ Barras de herramientas como botones
- ❑ Listas desplegables
- ❑ Y mucho más...

Las aplicaciones son conducidas por **eventos** y se desarrollan haciendo uso de las clases que nos ofrece la API de Java.

Interfaces gráficas de usuario (GUI)

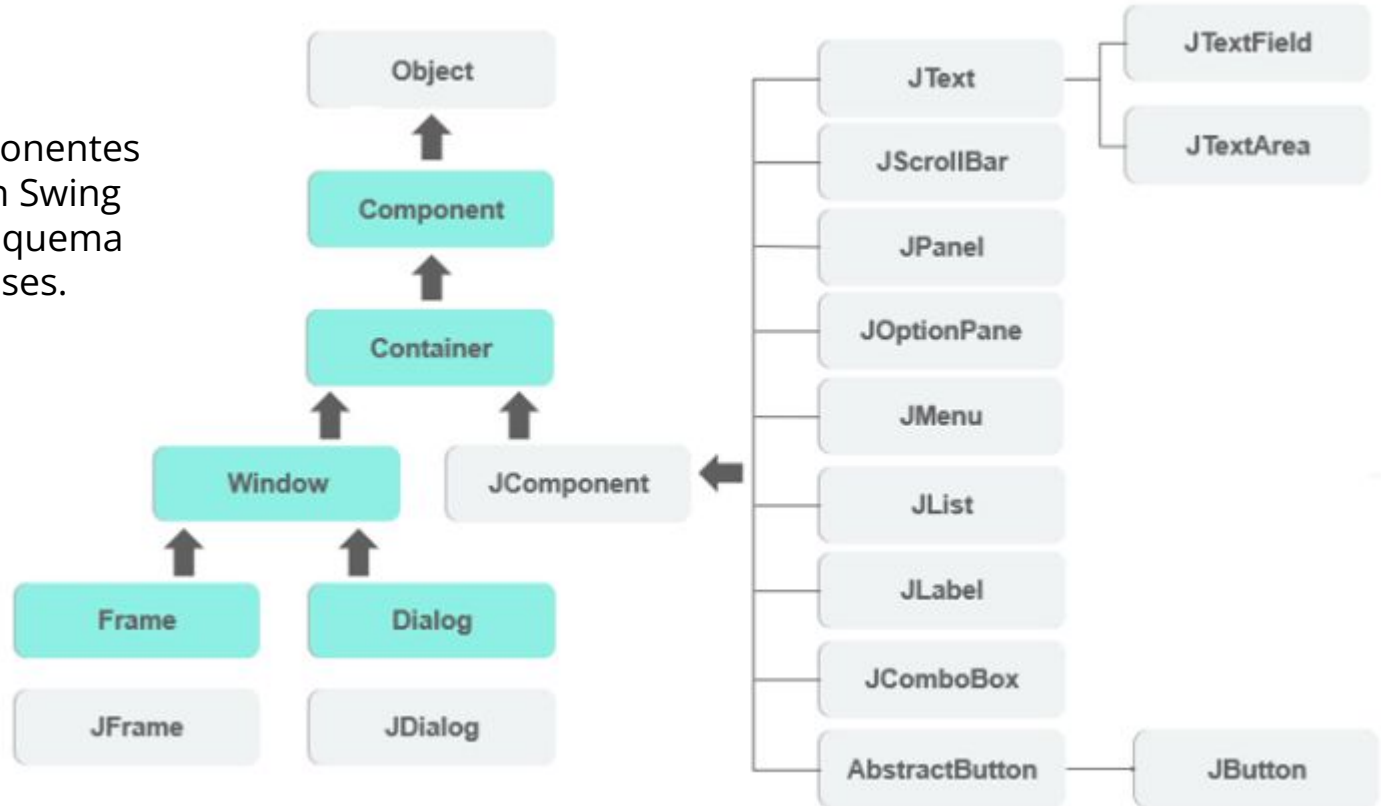
Cuando apareció Java, los componentes gráficos disponibles para el desarrollo GUI se encontraban en una biblioteca conocida como **Abstract Window Toolkit** (AWT).

AWT es adecuada para interfaces gráficas sencillas, pero no para proyectos más **complejos**.

Al aparecer Java 2, la biblioteca AWT se reemplazó por otra más robusta, versátil y flexible denominada **Swing**. A pesar de ello, AWT sigue estando disponible e incluso es usada por algunos componentes de Swing.

Swing

Algunos de los componentes más importantes en Swing mostrados en un esquema jerárquico de clases.



AWT vs. Swing

AWT

Plataforma activa: utiliza componentes gráficos nativos del sistema operativo en el que se ejecuta la aplicación, lo que significa que la apariencia de la GUI se adapta al aspecto nativo del sistema. Esto puede ser beneficioso para la integración con el sistema operativo, pero puede resultar en diferencias de apariencia entre plataformas.

Rendimiento: puede ofrecer un mejor rendimiento en algunas situaciones, especialmente cuando se trata de aplicaciones simples. Sin embargo, esto puede variar según la plataforma y la complejidad de la GUI.

Limitaciones de estilo: tiene un conjunto limitado de componentes de interfaz de usuario, lo que puede dificultar la creación de interfaces de usuario altamente personalizadas.

Swing

Basado en Java: está completamente escrito en Java y no depende de componentes nativos del sistema operativo. Esto asegura la consistencia en la apariencia y el comportamiento en todas las plataformas.

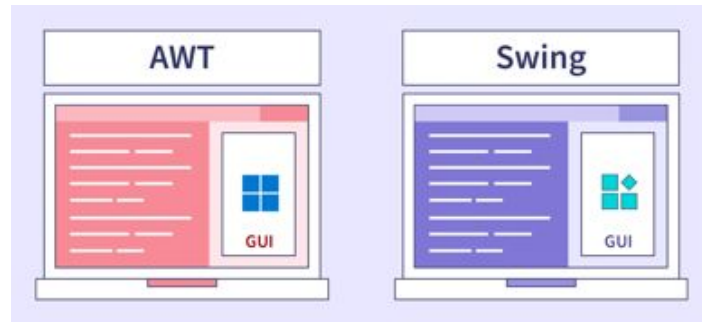
Personalización: ofrece una amplia gama de componentes de interfaz de usuario personalizables y altamente flexibles. Esto te permite crear interfaces de usuario más ricas y personalizadas.

Más pesado: ya que es una biblioteca puramente en Java, puede ser un poco más pesado en términos de consumo de recursos en comparación con AWT.

AWT vs. Swing

En resumen, la elección entre AWT y Swing depende de tus **necesidades** y **objetivos** específicos. Puedes consultar la **Figura 2.5** del libro para ver más diferencias.

- ❑ Si buscas una interfaz de usuario **simple** y deseas aprovechar las características **nativas** del sistema operativo, AWT puede ser adecuado.
- ❑ Si necesitas una interfaz de usuario más **personalizable** y consistente en **todas las plataformas**, Swing es una mejor opción.



Java

Swing

Swing

- JComponent: superclase de todos los componentes de Swing.
- JFrame: ventana que no está contenida en otras ventanas.
- JDialog: cuadro de diálogo.
- JPanel: contenedor invisible que mantiene componentes de interfaz y que se puede anidar, colocándose en otros paneles o en ventanas. También sirve de lienzo.

Categorías de clases

Contenedores

JFrame
JApplet
JWindow
JDialog

...

Categorías de clases

Componentes intermedios

JPanel
JScrollPane
...

Contenedores

JFrame
JApplet
JWindow
JDialog
...

Categorías de clases

Componentes intermedios

JPanel
JScrollPane
...

Contenedores

JFrame
JApplet
JWindow
JDialog
...

Componentes

JLabel
JButton
JTextField
JTextArea
...

Categorías de clases

Clases de soporte

Graphics
Color
Font
...

Componentes intermedios

JPanel
JScrollPane
...

Contenedores

JFrame
JApplet
JWindow
JDialog
...

Componentes

JLabel
JButton
JTextField
JTextArea
...

Ejemplo GUI, Ventana 1

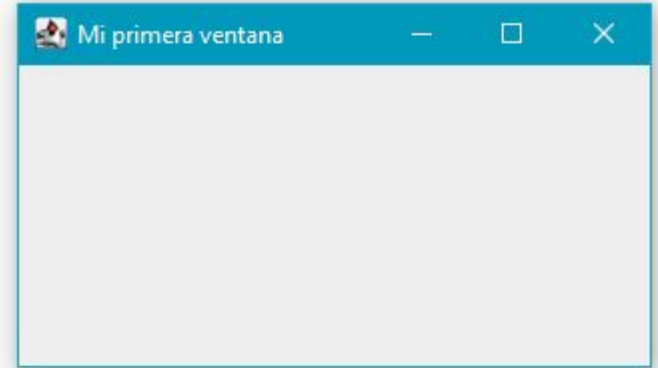
En Swing las ventanas están representadas por la clase `JFrame`. Se puede crear una de estas ventanas en el constructor.

```
public class Ventana1 extends JFrame {  
    /**  
     * Constructora que recibe el título de la ventana  
     * @param _msg  
     */  
    public Ventana1(String _msg) {  
        super(_msg);  
    }  
}
```

Se puede establecer su tamaño y localización.

```
this.setLocation(50, 100);  
this.setSize(400, 200);
```

Ejemplo GUI, Ventana 1



Hacer que sea visible.

```
this.setVisible(true);
```

Y finalizar la aplicación al cerrar la ventana.

```
this.setDefaultCloseOperation(EXIT_ON_CLOSE);
```

Java

Contenedores

Añadiendo componentes a la ventana

Toda aplicación Swing tiene, **al menos**, una ventana.

Cada ventana tiene asociado un panel: `ContentPane`.

Sobre él se dibujan los componentes y los gráficos.

El `ContentPane` es el **contenedor** de componentes. Estos componentes se añaden usando el método `getContentPane`.

```
private Container panelPrincipal;  
this.panelPrincipal = this.getContentPane();
```

Disposición de los componentes

Los componentes se agregan usando el método add.

```
private JLabel etiqueta;  
this.etiqueta = new JLabel("Bienvenido a mi segundo conversor");  
this.panelPrincipal.add(this.etiqueta);
```

El efecto de add **depende** del esquema de colocación o disposición (layout) del contenedor que se use. Existen diversos esquemas:

- ❑ FlowLayout: un componente tras otro de izquierda a derecha.
- ❑ BorderLayout: cinco regiones en el contenedor (NORTH, SOUTH, ...).
- ❑ GridLayout: contenedor en filas y columnas.
- ❑ GridBagLayout: contenedor por celdas.

FlowLayout

- Los componentes se van colocando de izquierda a derecha.
- Cuando una línea se completa se pasa a la siguiente.
- Mantiene el tamaño de los componentes.
- El constructor es `FlowLayout()`.
- Para organizar el contenedor se usa el método `setLayout()`.

Nota: consulta el **Punto 2.5.2** del libro para saber más.

BorderLayout

- Coloca los componentes en los cuatro puntos cardinales y el centro.
- El del centro se expande para ocupar el mayor área posible.
- El resto ocupan el menor espacio posible.
- El constructor es `BorderLayout()`.
- El método `add` tiene un parámetro adicional para indicar la zona (NORTH, SOUTH, WEST, EAST, CENTER).

Nota: consulta el **Punto 2.5.4** del libro para saber más.

BorderLayout

- Coloca los componentes en los cuatro puntos cardinales y el centro.
- El del centro se expande para ocupar el mayor área posible.
- El resto ocupan el menor espacio posible.
- El constructor es `BorderLayout()`.
- El método `add` tiene un parámetro adicional para indicar la zona (NORTH, SOUTH, WEST, EAST, CENTER).

Nota: consulta el **Punto 2.5.4** del libro para saber más.

GridLayout

- Parrilla de celdas iguales que se rellena de izquierda a derecha, línea a línea.
- Cambia el tamaño de los componentes.
- El número de filas y columnas se indica en el constructor (el otro parámetro se deja a 0).
- El Constructor es `GridLayout(filas, columnas)`.

Nota: consulta el **Punto 2.5.3** del libro para saber más.

GridBagLayout

- Organiza los componentes en una cuadrícula, donde cada celda de la cuadrícula puede contener un componente o estar vacía, haciendo que la disposición sea más flexible.
- Puedes especificar el tamaño, la alineación vertical y horizontal, el relleno (padding) y la expansión de los componentes de manera individual.
- permite superponer componentes en la misma celda, lo que puede ser útil para crear diseños más complejos.
- Los componentes pueden expandirse o contraerse automáticamente cuando se cambia el tamaño de la ventana, lo que facilita la creación de interfaces de usuario responsive.

Tarea: busca los métodos más utilizados con el contenedor GridBagLayout (constructoras, coordenadas de la celda, relleno, etc.).

Nota: consulta el **Punto 2.5.5** del libro para saber más.

Java

Componentes

Tipos de componentes

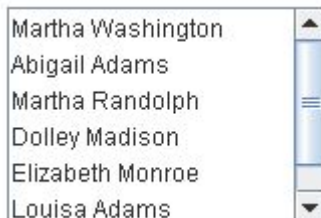
Leer el **Punto 2.6** del libro.

Un esquema muy interesante para tener una idea de la herencia presente en los componentes de Java es la **Figura 3.1** del libro.

Tarea: echa un vistazo al código proporcionado por el profesor (generado con chatGPT) y trata de entender todas las líneas del código. Ejecútalo y mira si el resultado es el que esperabas. Además, dibuja un perfil de la ventana, ya que esto te ayudará a entender mucho mejor cada una de las capas.

Más componentes Swing

Existen muchos componentes Swing que pueden ser usados:



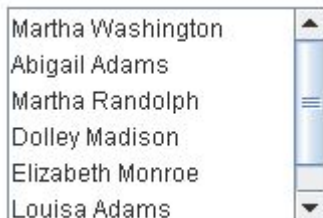
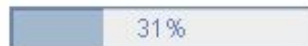
Host	User	Password	Last Modified
Biocca Games	Freddy	!#asf6Awwzb	Mar 16, 2006
zabble	ichabod	Tazb!34\$fZ	Mar 6, 2006
Sun Developer	fraz@hotmail.co...	AasW541!fbZ	Feb 22, 2006
Heirloom Seeds	shams@gmail....	bkz[ADF78!	Jul 29, 2005
Pacific Zoo Shop	seal@hotmail.c...	vbAf124%z	Feb 22, 2006

Más componentes Swing

Existen muchos componentes Swing que pueden ser usados:



<https://web.mit.edu/6.005/www/sp14/psets/ps4/java-6-tutorial/components.html>



Host	User	Password	Last Modified
Biocca Games	Freddy	!#asf6Awwzb	Mar 16, 2006
zabble	ichabod	Tazb!34\$fZ	Mar 6, 2006
Sun Developer	fraz@hotmail.co...	AasW541!fbZ	Feb 22, 2006
Heirloom Seeds	shams@gmail....	bkz[ADF78!	Jul 29, 2005
Pacific Zoo Shop	seal@hotmail.c...	vbAf124%z	Feb 22, 2006

Java

Ventanas de ejemplo

Ejemplo GUI, Ventana 2

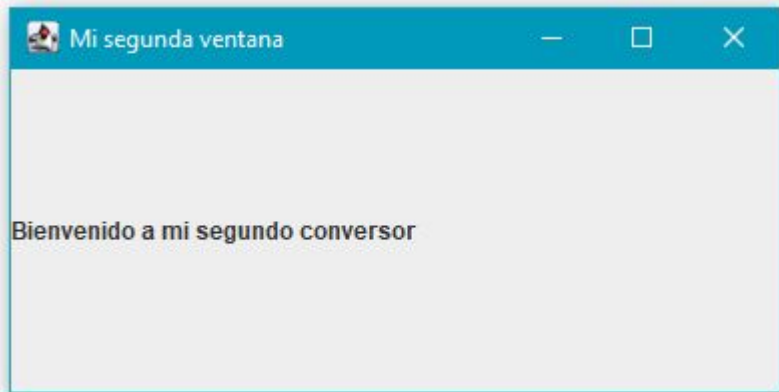
Prueba el siguiente código:

```
public Ventana2(String _msg) {  
    super(_msg);  
    // Contenedor principal  
    this.panelPrincipal = this.getContentPane();  
    this.etiqueta = new JLabel("Bienvenido a mi segundo conversor");  
    // Inserto el componente en el contenedor  
    this.panelPrincipal.add(this.etiqueta);  
    // Establezco el resto de propiedades de la ventana  
    this.setLocation(50, 100);  
    this.setSize(400, 200);  
    this.setVisible(true);  
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);  
}
```

Ejemplo GUI, Ventana 2

Prueba el siguiente código:

```
public Ventana2(String _msg) {  
    super(_msg);  
    // Contenedor principal  
    this.panelPrincipal = this.getContentPane();  
    this.etiqueta = new JLabel("Bienvenido a mi segundo conversor");  
    // Inserto el componente en el contenedor  
    this.panelPrincipal.add(this.etiqueta);  
    // Establezco el resto de propiedades de la ventana  
    this.setLocation(50, 100);  
    this.setSize(400, 200);  
    this.setVisible(true);  
    this.setDefaultCloseOperation(EXIT_ON_CLOSE);  
}
```



Ejemplo GUI, Ventana 3

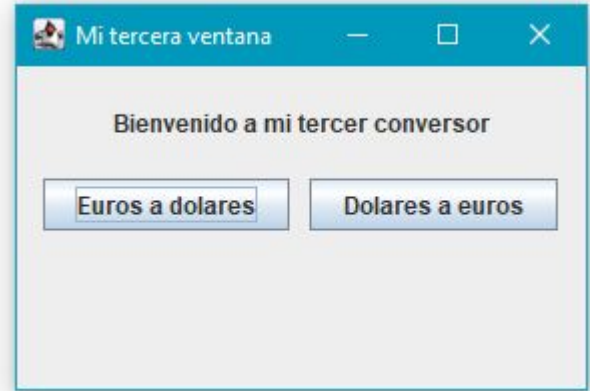
Añade las siguientes líneas al ejemplo anterior:

```
// Creo los botones
JButton euroADolar = new JButton("Euros a dolares");
JButton dolarAEuro = new JButton("Dolares a euros");
// Inserto los elementos en el contenedor principal
this.panelPrincipal.add(this.etiqueta);
this.panelPrincipal.add(euroADolar);
this.panelPrincipal.add(dolarAEuro);
```

Ejemplo GUI, Ventana 3

Añade las siguientes líneas al ejemplo anterior:

```
// Creo los botones
JButton euroADolar = new JButton("Euros a dolares");
JButton dolarAEuro = new JButton("Dolares a euros");
// Inserto los elementos en el contenedor principal
this.panelPrincipal.add(this.etiqueta);
this.panelPrincipal.add(euroADolar);
this.panelPrincipal.add(dolarAEuro);
```



Ejemplo GUI, Ventana 4

Añade las siguientes líneas al ejemplo anterior:

```
// Configuro el panel norte
this.panelNorte = new JPanel();
this.panelNorte.setLayout(new FlowLayout());
this.etiqueta = new JLabel("Bienvenido a mi cuarto conversor");
this.panelNorte.add(this.etiqueta);
// Configurar el panel sur
// Inserto el panel norte en el contenedor principal
this.panelPrincipal.add(this.panelNorte, BorderLayout.NORTH);
// Insertar el panel sur en el contenedor principal
```

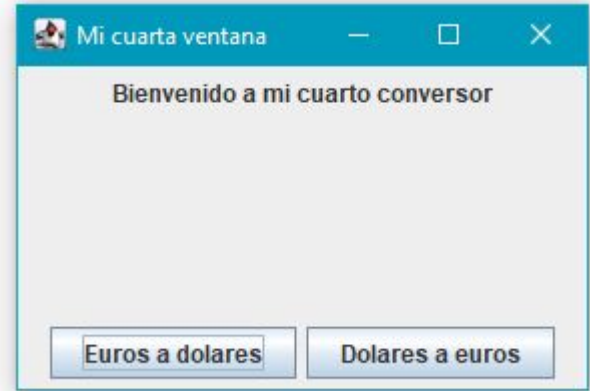
Haz las tareas comentadas en azul.

Ejemplo GUI, Ventana 4

Añade las siguientes líneas al ejemplo anterior:

```
// Configuro el panel norte
this.panelNorte = new JPanel();
this.panelNorte.setLayout(new FlowLayout());
this.etiqueta = new JLabel("Bienvenido a mi cuarto conversor");
this.panelNorte.add(this.etiqueta);
// Configurar el panel sur
// Inserto el panel norte en el contenedor principal
this.panelPrincipal.add(this.panelNorte, BorderLayout.NORTH);
// Insertar el panel sur en el contenedor principal
```

Haz las tareas comentadas en azul.



Ejemplo GUI, Ventana 5

Añade los siguientes atributos:

```
// Insertamos campos de texto y etiquetas en un nuevo panel central
private JPanel panelCentral;
private JLabel pedirCambio;
private JTextField cambio;
private JLabel pedirCantidad;
private JTextField cantidad;
```

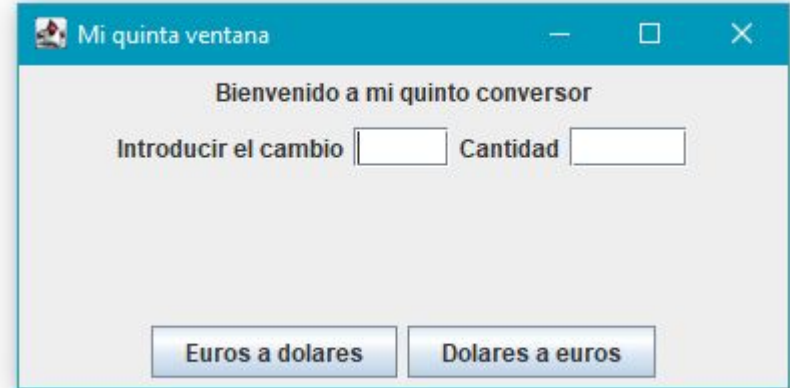
Inicializa los nuevos elementos e insértalos en el panel central. Finalmente, añade el panel central al contenedor principal.

Ejemplo GUI, Ventana 5

Añade los siguientes atributos:

```
// Insertamos campos de texto y etiquetas en un nuevo panel central
private JPanel panelCentral;
private JLabel pedirCambio;
private JTextField cambio;
private JLabel pedirCantidad;
private JTextField cantidad;
```

Inicializa los nuevos elementos e insértalos en el panel central. Finalmente, añade el panel central al contenedor principal.



Java

Programación por eventos

Programación dirigida por eventos

Antes de ver esta sección, vamos a leer el **Punto 3.1** y **3.2** del libro.

Un evento es una acción **relevante** para la aplicación. Por ejemplo, las pulsaciones del teclado o ratón, el vencimiento de un temporizador, etc.

Los programas atiende a los **sucesos** que ocurren y, dependiendo de cuáles sean, ejecutan unas **funciones** u otras. No existe un **único** flujo de ejecución.

Por lo tanto, para programar lo que hacemos es especificar los **eventos** a los que debe responder el programa y, al mismo tiempo, las **acciones** a realizar cuando ocurran dichos eventos.

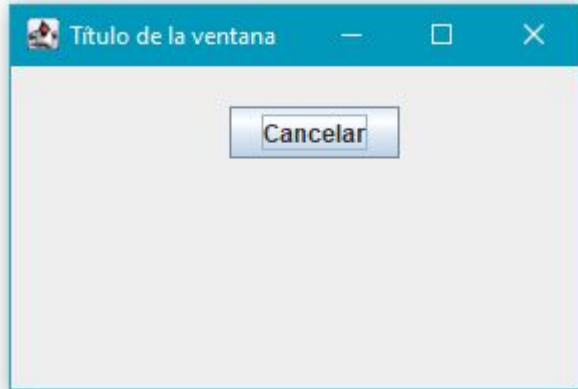
Interacción con el usuario

Cuando el usuario de un programa mueve el ratón, presiona un pulsador o pulsa una tecla, genera un evento `ActionEvent`.

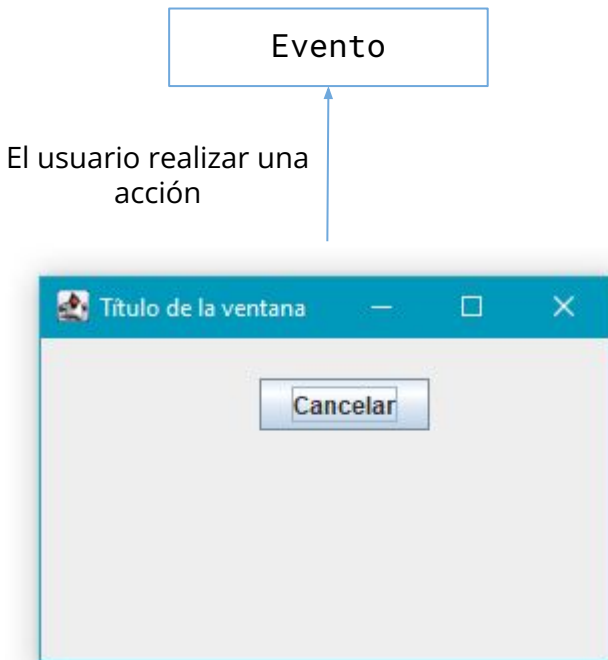
Los eventos son objetos de ciertas clases que indican:

- El elemento que accionó el usuario.
- La identificación del evento que indica la naturaleza del evento.
- La posición del ratón en el momento de la interacción.
- Teclas adicionales pulsadas por el usuario como la tecla de Control, cambio de mayúsculas a minúsculas, etc.

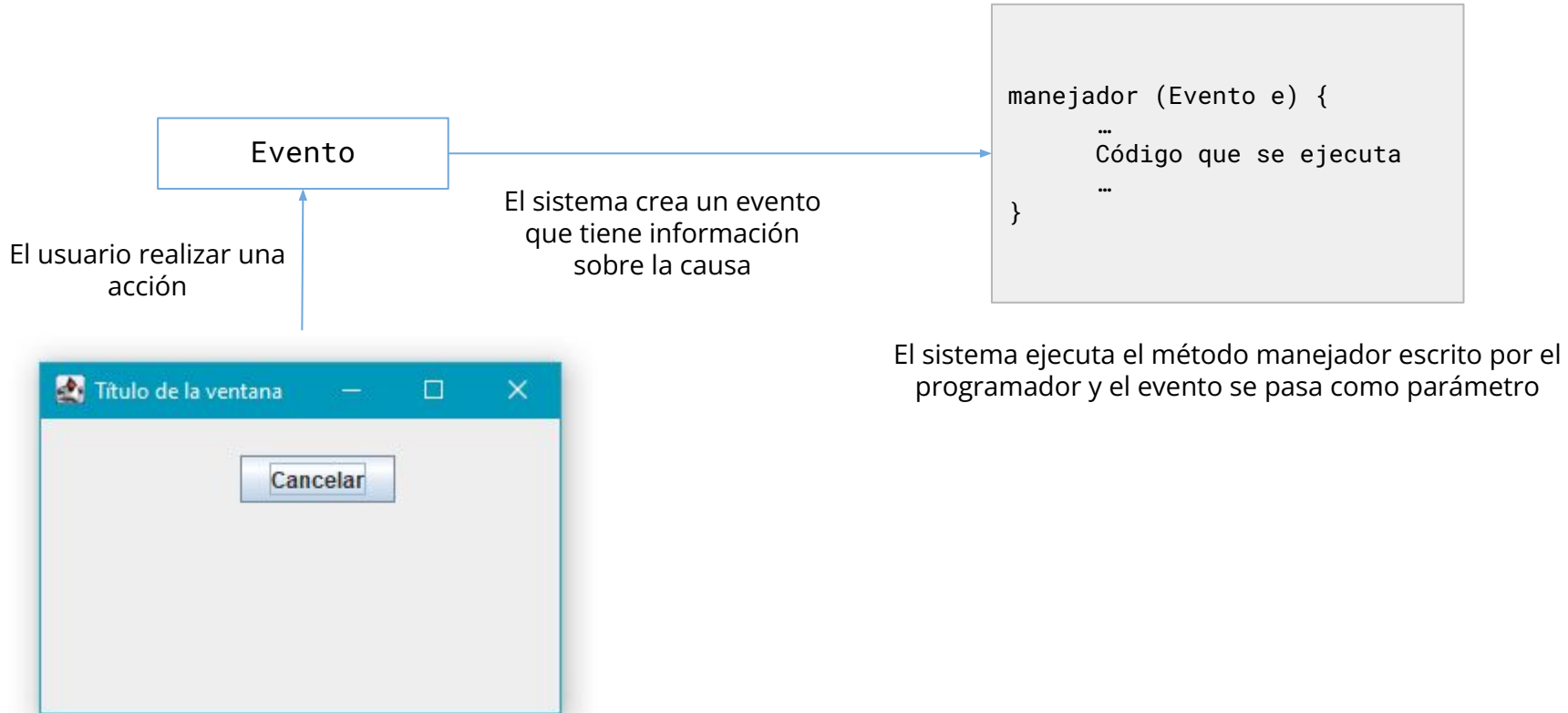
Generación de eventos



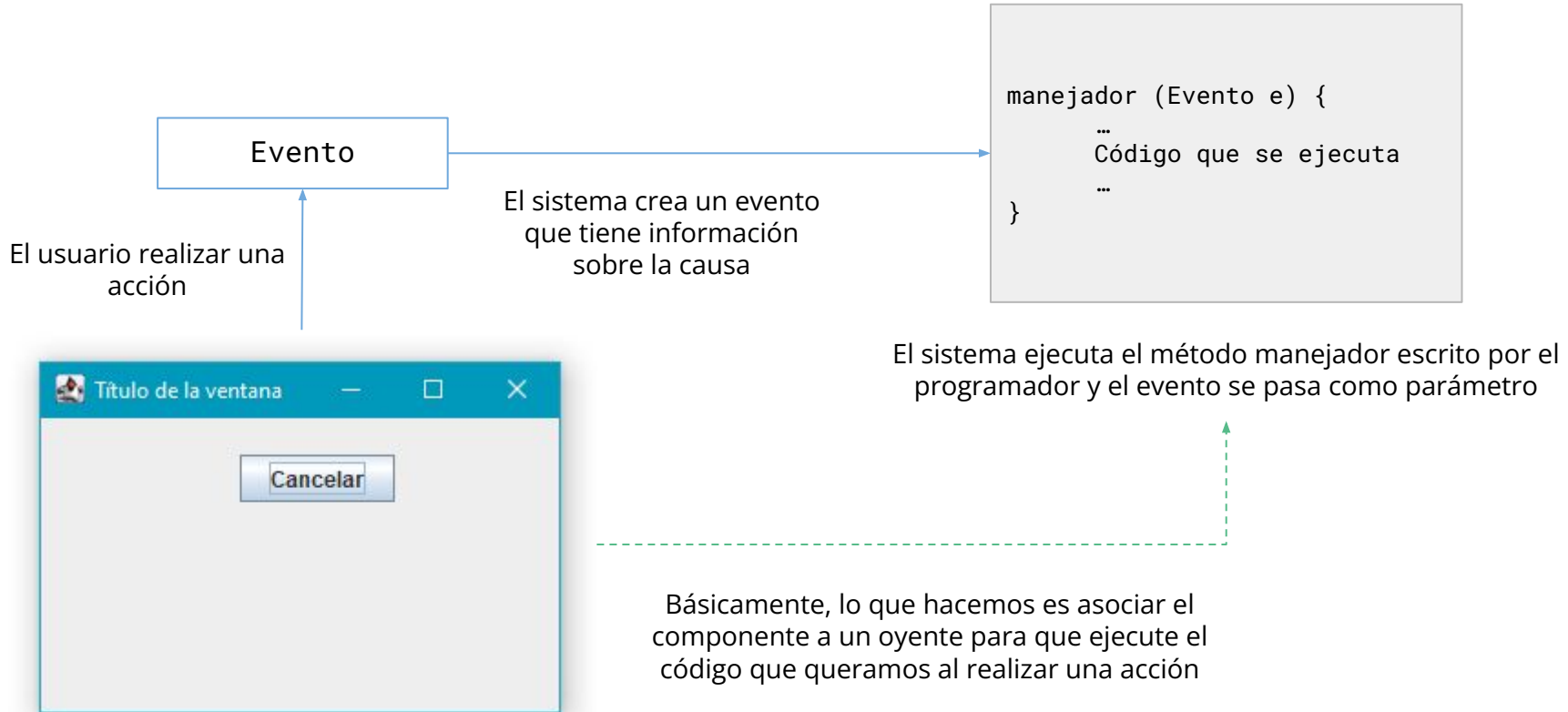
Generación de eventos



Generación de eventos



Generación de eventos



Java

Programación por eventos

Definición

Un ActionListener es una interfaz que pertenece al paquete `java.awt.event`. Es parte del modelo de eventos de Java, utilizado en aplicaciones gráficas para **responder a acciones** específicas.

Los ActionListener permiten que los objetos "escuchen" eventos de acción, reaccionando a ellos cuando ocurren. Por esta razón, son particularmente útiles cuando se desea ejecutar un código específico en **respuesta a una acción** del usuario, como hacer clic en un botón, seleccionar un elemento en un menú o presionar una tecla.

¿Por qué son importantes los ActionListener?

No es que sean importantes, es que directamente son **fundamentales** en el desarrollo de interfaces gráficas interactivas.

Permiten que una aplicación responda a las acciones del usuario de manera dinámica, haciendo que las interfaces sean más interactivas y útiles.

Son esenciales para gestionar eventos y diseñar aplicaciones que respondan a la interacción del usuario, creando una experiencia de usuario fluida y reactiva.

Genera un ejemplo I

Tarea: utiliza alguna IA para generar un ejemplo sencillo en el que se utilice un ActionListener.

Supongamos el siguiente ejemplo:

```
button.addActionListener(new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        // Acción a realizar cuando se hace clic en el botón  
        System.out.println("¡Botón presionado!");  
    }  
});
```

El evento que se captura es de tipo `ActionEvent` que, como ves, es lo que estamos recibiendo como parámetro de entrada `e`. Este ejemplo es sencillo pero, entre otras cosas, podríamos identificar el botón que se ha pulsado (`e.getSource()`) o el instante en el que ha sido pulsado (`e.getWhen()`), entre otros.

Genera un ejemplo II

Para que te hagas una idea, el flujo que se sigue viene a ser algo como:

1. El método `actionPerformed` está siempre escuchando cualquier evento que de produzca.
2. Supongamos que, ahora, el usuario pulsa la tecla A.
3. En ese instante, el método `actionPerformed` recibe la `e`, que es una variable que contiene toda la información con el tipo de evento que se ha producido (no es lo mismo que se pulse una tecla, que se haga clic, que se haga hover en una ventana, etc.).
4. Una vez que tenemos el evento almacenado en `e`, entonces podemos “jugar” con él. Por ejemplo, dentro del método podríamos tener un `if` del estilo: si se ha pulsado la tecla A, entonces muestra un mensaje que diga: *la tecla A ha sido pulsada*. Si se pulsa cualquier otra tecla, entonces muestra un mensaje que diga: *otra tecla que no es la A ha sido pulsada*.

Tarea: crea una ventana que permita distinguir alguna teclas pulsadas y muestre mensajes.

Ejemplo ActionListener

Hagamos que suene un pitido al pulsar el botón. En primer lugar hay que crear la clase oyente para que implemente la interfaz `ActionListener`. Y en segundo lugar, asociar la clase oyente con el botón mediante el método `addActionListener()`.

```
public class OyenteEurosADolar implements ActionListener {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        java.awt.Toolkit.getDefaultToolkit().beep();  
    }  
}
```

Ejemplo ActionListener

Hagamos que suene un pitido al pulsar el botón. En primer lugar hay que crear la clase oyente para que implemente la interfaz `ActionListener`. Y en segundo lugar, asociar la clase oyente con el botón mediante el método `addActionListener()`.

```
public class OyenteEurosADolar implements ActionListener {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        java.awt.Toolkit.getDefaultToolkit().beep();  
    }  
}
```



```
this.euroADolar.addActionListener(new OyenteEurosADolar());
```

Ejemplo ActionListener con clases anónimas

En lugar de crear una clase por cada tipo de evento que se quiere gestionar, la mayoría de las veces usamos clases anónimas. Nótese que siempre que aparezca un **new**, es que se está instanciando algo (en este caso, una clase anónima).

```
// Implementación ActionListener
this.euroADolar.addActionListener(
    new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            java.awt.Toolkit.getDefaultToolkit().beep();
        }
    });
```

Añade este código al botón indicado y
comprueba el resultado.

Ejercicios

Tarea: realiza los tres ejercicios propuestos por el profesor. Debes simular el mismo comportamiento que cada una de las tres ventanas.

Componentes Visuales



Curso 2024/25