

Unidad 3 : Creación de componentes visuales

1.- Concepto de componente. Características.

Un **componente software** es una clase creada para ser reutilizada y que puede ser manipulada por una herramienta de desarrollo de aplicaciones visual. Se define por su **estado** que se almacena en un conjunto de **propiedades**, las cuales pueden ser modificadas para adaptar el componente al programa en el que se inserte. También tiene un **comportamiento** que se define por los eventos ante los que responde y los **métodos** que ejecuta ante dichos eventos.

Un subconjunto de los atributos y los métodos forman la **interfaz** del componente.

Para que pueda ser distribuida se empaqueta con todo lo necesario para su correcto funcionamiento, quedando independiente de otras bibliotecas o componentes.

Para que una clase sea considerada un componente debe cumplir ciertas normas:

- Debe poder modificarse para adaptarse a la aplicación en la que se integra.
- Debe tener persistencia, es decir, debe poder guardar el estado de sus propiedades cuando han sido modificadas.
- Debe tener introspección, es decir, debe permitir a un IDE que pueda reconocer ciertos elementos de diseño como los nombres de las funciones miembros o métodos y definiciones de las clases, y devolver esa información.
- Debe poder gestionar **eventos**.

El desarrollo basado en componentes tiene, además, las siguientes **ventajas**:

- Es mucho más sencillo, se realiza en menos tiempo y con un coste inferior.
- Se disminuyen los errores en el software ya que los componentes se deben someter a un riguroso control de calidad antes de ser utilizados.

Autoevaluación

Imagina una herramienta de desarrollo, están compuestas de un conjunto de ventanas que podemos mostrar u ocultar, cambiar su tamaño, etc., de modo que al cerrar la aplicación y volverla a abrir se mantiene la estructura que tenía al cerrarse. ¿Con qué característica de los componentes relacionas esto?

- ☒ Con la persistencia.
- ☐ Con la introspección.
- ☐ Con la gestión de eventos.

Así es, la persistencia, que permite que los cambios en las propiedades de las ventanas se mantengan de una sesión a otra.

2.- Elementos de un componente: propiedades y atributos.

Como en cualquier clase, un componente tendrá definido un estado a partir de un conjunto de **atributos**. Los atributos son variables definidas por su nombre y su tipo de datos que toman valores concretos. Normalmente los atributos son privados y no se ven desde fuera de la clase que implementa el componente, se usan sólo a nivel de programación.

Las propiedades son un tipo específico de atributos que representan características de un componente que afectan a su apariencia o a su comportamiento. Son accesibles desde fuera de la clase y forman parte de su interfaz. Suelen estar asociadas a un atributo interno.

Las propiedades de un componente pueden examinarse y modificarse mediante métodos o funciones miembro, que acceden a dicha propiedad, y pueden ser de dos tipos:



- **getter:** permiten leer el valor de la propiedad. Tienen la estructura:

```
public <TipoPropiedad> get<NombrePropiedad>( )
```

si la propiedad es booleana el método getter se implementa así:

```
public boolean is<NombrePropiedad>()
```

- **setter:** permiten establecer el valor de la propiedad. Tiene la estructura:

```
public void set<NombrePropiedad>(<TipoPropiedad> valor)
```

Si una propiedad no incluye el método set entonces es una propiedad de **sólo lectura**.

Por ejemplo, si estamos generando un componente para crear un botón circular con sombra, podemos tener, entre otras, una propiedad de ese botón que sea **color**, que tendría asociados los siguientes métodos:

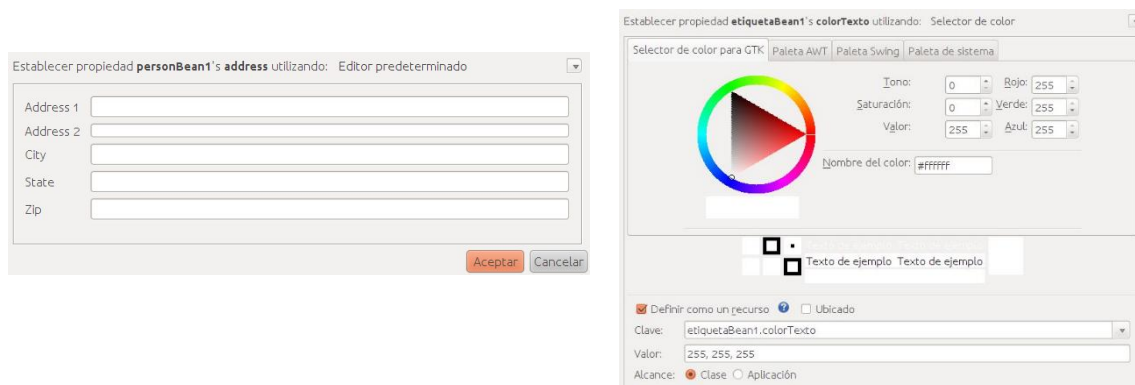
```
public void setColor(String color)
```

```
public String getColor()
```

2.1.- Modificar gráficamente el valor de una propiedad con un editor.

Una de las principales características de un componente es que una vez instalado en un entorno de desarrollo, éste debe ser capaz de identificar sus propiedades simplemente detectando parejas de operaciones get/set, mediante la capacidad denominada **introspección**.

El entorno de desarrollo podrá editar automáticamente cualquier propiedad de los tipos básicos o de las clases **Color** y **Font**. Aunque no podrá hacerlo si el tipo de datos de la propiedad es algo más complejo, por ejemplo, si usamos otra clase, como **Cliente**. Para poder hacerlo tendremos que crear nuestro propio editor de propiedades, por ejemplo, en la primera imagen puedes ver un editor programado para un componente que almacena información de personas, el nombre o el teléfono son cadenas de caracteres que se editan fácilmente, pero la dirección es una propiedad compuesta que precisa del siguiente editor. En la segunda imagen aparece el editor de una propiedad de tipo **Color**.



Un **editor de propiedad** es una herramienta para personalizar un tipo de propiedad en particular. Los editores de propiedades se utilizan en la ventana **Propiedades**, que es donde se determina el tipo de la propiedad, se busca un editor de propiedades apropiado, y se muestra el valor actual de la propiedad de una manera adecuada a la clase.

La creación de un editor de propiedades usando tecnología Java supone programar una clase que implemente la interfaz **PropertyEditor**, que proporciona métodos para especificar cómo se debe mostrar una propiedad en la hoja de propiedades. Su nombre debe ser el nombre de la propiedad seguido de la palabra **Editor**:

```
public <Propiedad>Editor implements PropertyEditor {...}
```

Por defecto la clase **PropertyEditorSupport** que implementa **PropertyEditor** proporciona los editores más comúnmente empleados, incluyendo los mencionados tipos básicos, **Color** y **Font**.

Una vez que tengamos todos los editores, tendremos que empaquetar las clases con el componente para que use el editor que hemos creado cada vez que necesite editar la propiedad. Así, conseguimos que cuando se añada un componente en un panel y lo seleccionemos, aparezca una hoja de propiedades, con la lista de las propiedades del componente y sus editores asociados para cada una de ellas. El IDE llama a los métodos **getters**, para mostrar en los editores los valores de las propiedades. Si se cambia el valor de una propiedad, se llama al método **setter**, para actualizar el valor de

dicha propiedad, lo que puede o no afectar al aspecto visual del componente en el momento del diseño.

Para saber más

Si quieres conocer un poco más de este apartado de la tecnología Java puedes consultar la página del tutorial que tiene Oracle alojado en su página (está en inglés) y la especificación de la interfaz **PropertyEditor**:

[Interfaz PropertyEditor.](#)

Autoevaluación

Si necesitamos crear un editor para una propiedad que se llame **dirección**, indica cuál es la manera correcta de definir la clase para programarlo:

```
private direccion implements PropertyEditor{...}.  
public direccionEditor {...}.  
public direccionEditor implements PropertyEditor {...}.
```

Así es, se debe definir la clase como pública, el nombre se forma mediante el nombre de la propiedad seguido de la palabra Editor y, además, debe implementar la interfaz **PropertyEditor**.

2.1.1.- Ejemplo de creación de un componente con un editor de propiedades.

Los editores de propiedades tienen dos propósitos fundamentalmente:

- Convertir el valor de y a cadenas para ser mostrados adecuadamente conforme a las características de la propiedad, y
- Validar los datos nuevos cuando son introducidos por el usuario.

Los pasos básicos para crear un editor de propiedades consisten en:

1. Crear una clase que extienda a **PropertyEditorSupport**.
2. Añadir los métodos **getAsText** y **setAsText**, que transformarán el tipo de dato de la propiedad en cadena de caracteres o viceversa.
3. Añadir el resto de métodos necesarios para la clase.
4. Asociar el editor de propiedades a la propiedad en cuestión.

Para ilustrar como se implementa un componente con una herramienta como NetBeans, adaptaremos un campo de texto de manera que podamos modificar mediante propiedades el **ancho** que ocupa en el formulario (número de columnas), el **color** del texto y la **fuentes**. De esta manera veremos como modificar desde NetBeans propiedades sencillas que no necesitan un editor. Además, vamos a añadir una propiedad, denominada **tipo**, que aunque es una cadena de caracteres requiere de un editor de propiedades porque solo puede tomar tres posibles valores que se seleccionarán mediante una lista desplegable. Esta última propiedad contiene un tipo de dato, de manera que sólo se puedan escribir valores enteros, reales o Texto en el campo de texto. Finalmente tendremos las siguientes propiedades:

- El **ancho** que se puede representar como una propiedad de tipo entero.
- El **color** que será una propiedad de tipo **color**, que como hemos dicho cuenta con su propio editor.

- La **fuentes** será de tipo **Font**, y también cuenta con su propio editor.
- El **tipo** datos será una propiedad compuesta por el tipo de datos y el contenido del campo de texto en si que necesitará que programemos un editor de propiedades para el.

Veamos como crear este ejemplo.

Debes conocer

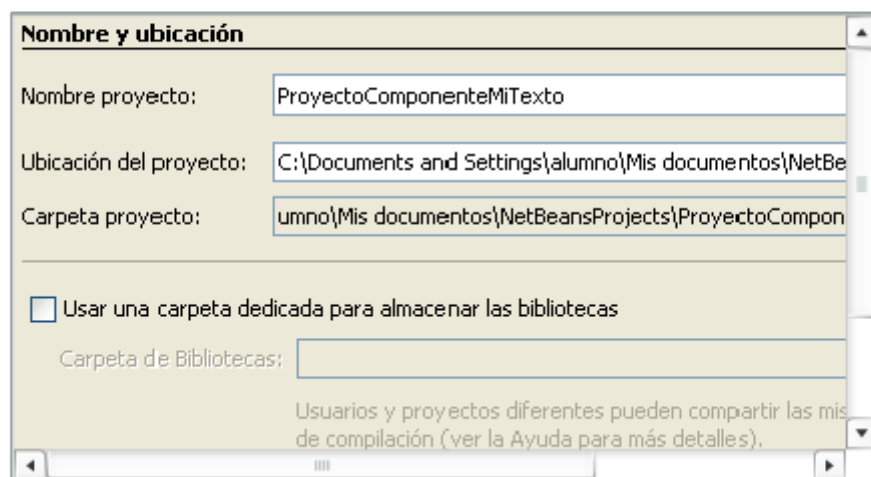
La siguiente presentación te indicará los pasos que debes seguir para crear el campo de texto que acabamos de describir:

Creación de un componente con un editor de propiedades



Primer paso: Crear el proyecto

Crear un proyecto de tipo Java Application con las opciones Crear clase principal y Configurar como proyecto principal desmarcadas. Llamar al proyecto ProyectoComponenteMiTexto



Añadir el componente

Añadir al proyecto un archivo de tipo Componente JavaBean pertenece a la categoría Objetos JavaBean accesible desde Archivo.

Llamar al archivo ComponenteMiTexto y ubicarlo en el paquete misControles

Nombre y ubicación

Nombre de Clase:

Proyecto:

Ubicación:

Heredar de un campo de texto

```
public class ComponenteMiTexto extends JTextField implements Serializable
```

El componente que estamos creando es una etiqueta modificada, por lo que heredaremos de la clase JTextField.

Añadir las propiedades

```
import java.beans.*;
import java.io.Serializable;

/**
 *
 * @author alumno
 */
public class ComponenteMiTexto implements Serializable {

    public ComponenteMiTexto() {
    }

}
```

NetBeans genera código, pero nosotros lo eliminaremos y
dejar el esqueleto básico de la clase.

Añadir propiedades

Nombre: =

Tipo:

☒ private ☐ package ☐ protected ☐ public

☐ static ☐ final

☒ Crear métodos getter y setter ☐ Generar getter ☐ Generar setter

☒ Generar Javadoc

☐ Delimitado

☐ Vetable

☐ Indexado

☐ Generar soporte para cambio de propiedad ☐ Generar soporte para...

Vista previa:

Agregar propiedad

Usando este método
agregamos las
propiedades ancho
(int), color (Color),
fuente (Font) y tipo
(String).

```
private Color color;  
  
/**  
 * Get the value of color  
 *  
 * @return the value of color  
 */  
public Color getColor() {  
    return color;  
}  
  
/**  
 * Set the value of color  
 *  
 * @param color new value of color  
 */
```

Crear un editor de propiedades

Añadir un editor de propiedades

- Un editor de propiedades personalizado se crea generando una interfaz gráfica que permita mostrar e introducir los datos de la propiedad, y
- Una clase espacial que deriva de PropertyEditorSupport que implementa el paso de la interfaz a la propiedad y viceversa.
- A continuación habrá que indicar al componente que la propiedad tiene un editor específico.



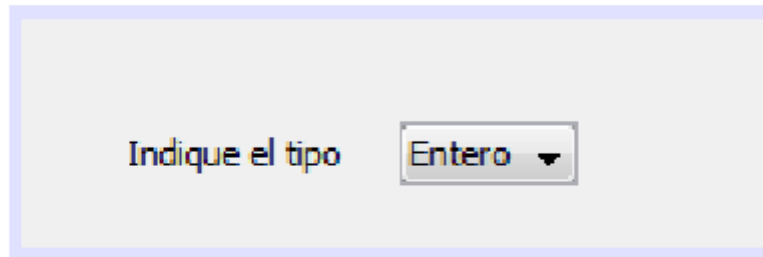
Primer paso: Crear el panel para la interfaz gráfica

Nombre y ubicación	
Nombre de Clase:	<input type="text" value="editorPanel"/>
Proyecto:	<input type="text" value="ProyectoComponenteMiTexto"/>
Ubicación:	<input type="text" value="Paquetes de fuentes"/>
Paquete:	<input type="text" value="misControles"/>
Crear Archivo:	<input type="text" value="nts\NetBeansProjects\ProyectoComponenteMiTexto\src\misControles\editorPanel.java"/>

Añade al proyecto un JPanel, que pertenezca al paquete misControles también, para editar en modo gráfico las propiedades del tipo. Dado que lo que necesitamos es que solo admita tres posibles valores vamos a añadir al panel una lista desplegable y una etiqueta..



Añadir el panel para editar la propiedad dirección



Se ha nombrado la lista desplegable como `cmbTipo` y se le ha asignado los valores Entero, Real, Texto a la propiedad `model`. Debes establecer la visibilidad de la lista a protegida o pública para que sea accesible desde el resto de las clases. Puedes hacerlo seleccionando la lista y en el menú contextual seleccionas Personalizar código.

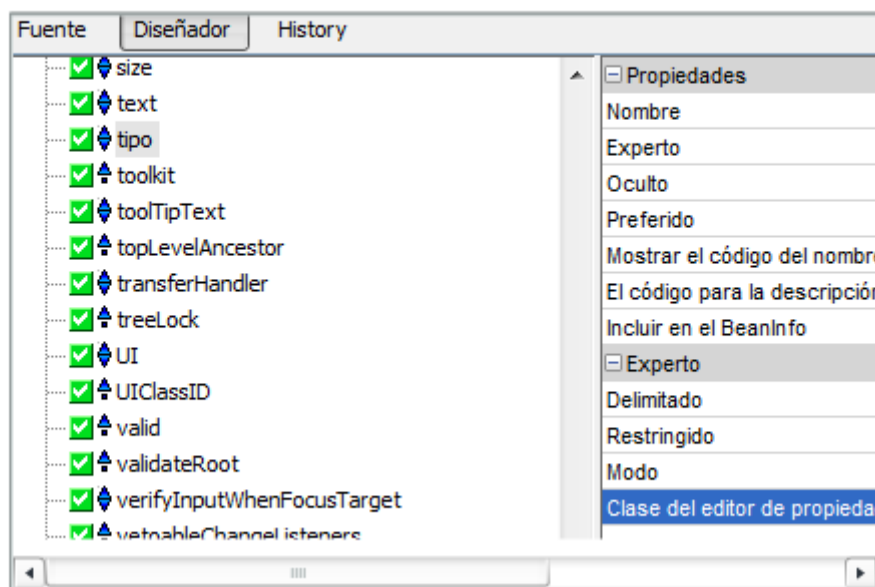
Crear el editor de propiedades

- Es una clase que extiende a `PropertyEditorSupport`.
- Permite asociar la propiedad tipo con la interfaz gráfica que has creado para recoger y mostrar su valor.
- Dispone como única propiedad de un editor de tipo `editorPanel`.
- Es preciso rellenar una serie de métodos como:
 - `getAsText` y `setAsText`
- Puedes encontrar el código completo de la clase en el proyecto que se adjunta con los contenidos.

Asociar el editor de propiedades con su propiedad

- En este caso lo más sencillo es crear un objeto de tipo **BeanInfo**.
- Para hacerlo haz clic con el botón secundario sobre la clase **ComponenteMiTexto** en el panel Proyectos y selecciona **Editor BeanInfo....** Te preguntará si quieres crear un BeanInfo nuevo, a lo que debes responder que si.
- Cuando lo hayas creado ábrelo con el Diseñador.
- Selecciona la propiedad tipo, en las propiedades que verás a la derecha encontrarás que puedes rellenar la clase del Editor de propiedades. Aquí debes poner la clase `misControles.tipoPropertyEditor.class`

Editor BeanInfo



Limpiar y Construir

Una vez que has creado todas las clases necesarias basta con que construyas el proyecto usando la opción **Limpiar y Construir** del menú contextual del proyecto.



Utilizar el componente

- Para usar el componente tienes que **añadirlo a la paleta**.
- Lo conseguirás si haces clic con el botón secundario sobre la clase **ComponenteMiTexto.java** en la opción Herramientas >> **Añadir a la Paleta**.
- Puedes elegir cualquier categoría, pero se recomienda hacerlo en **Beans Personalizados**.

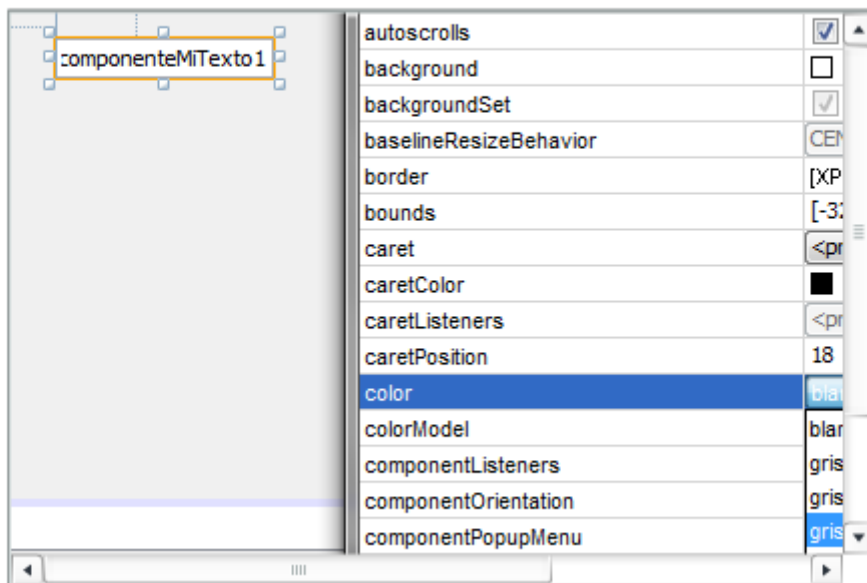


Utilizar el componente

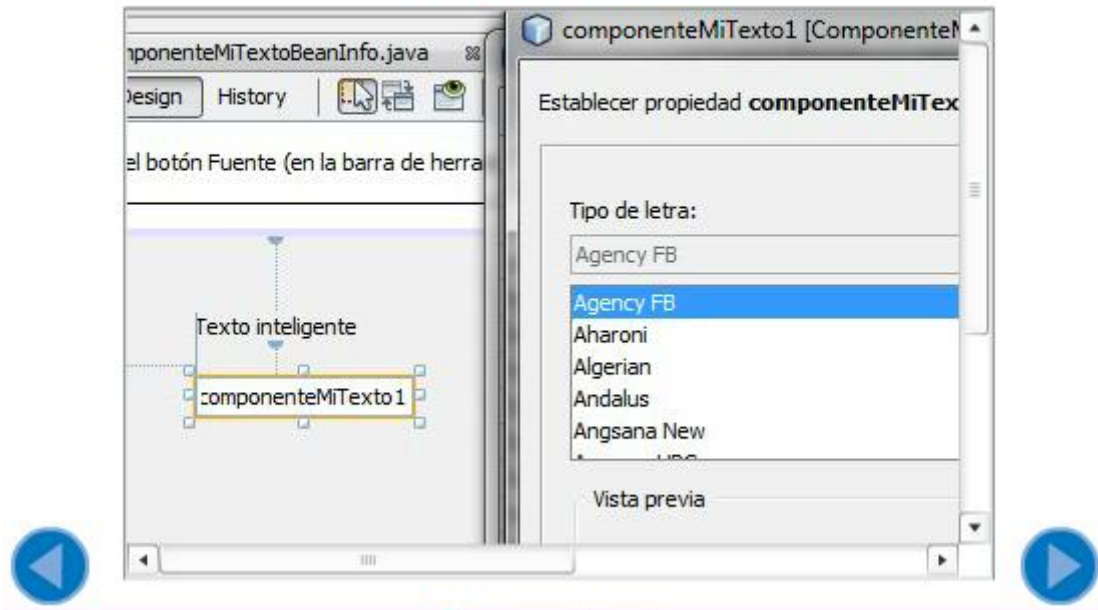
Si ahora creas un nuevo proyecto para una aplicación de escritorio podrás añadir desde la paleta de componentes, en el apartado Beans Personalizados un objeto de tipo `ComponenteMiTexto`.

Verás que si tratas de modificar la propiedad tipo desde el Inspector de propiedades se abre un panel para modificar el tipo.

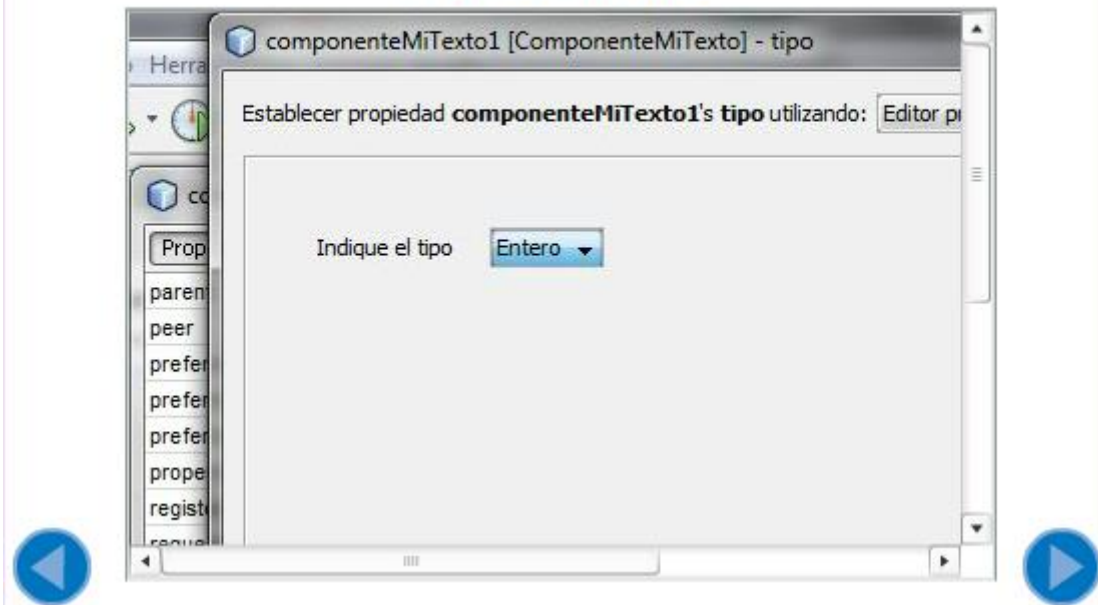
Propiedad color



Propiedad fuente



Propiedad tipo



Propiedades editables

- Si has observado las imágenes anteriores verás que según sea el tipo de la propiedad podemos cambiar el modo en que se rellena, los editores de **Color** y **Font** se cargan por defecto, y el editor de tipo lo hemos creado y cargado nosotros con una clase **PropertyEditor** y otra **BeanInfo**.
- Además de tener un editor personalizado, el tipo se puede escoger de una lista creada desde el método **setTags** de **PropertyEditorSupport**.



Terminar el componente

- La última fase de creación del componente sería gestionar, mediante código que los valores asignados a las propiedades tiene su efecto, es decir, que se cambia el color, la fuente y el ancho, y que el tipo es el adecuado. Para ello modificamos los métodos **setter** de las propiedades para asignar el valor a la propiedad original del campo de texto. Revisa el código que se entrega con la unidad.
- Más tarde se debería gestionar que cuando se escribiera en el campo de texto se cumpla que lo escrito coincide en tipo con el que esté seleccionado en ese momento. Debe hacerse mediante gestión de eventos y sobrescribiendo el método **setText**.



2.2.- Propiedades simples e indexadas.

- Una **propiedad simple** representa un único valor, un número, verdadero o falso o un texto por ejemplo.

Tiene asociados los métodos **getter** y **setter** para establecer y rescatar ese valor. Por ejemplo, si un componente de software tiene una propiedad llamada peso de tipo real susceptible de ser leída o escrita, deberá tener los siguientes métodos de acceso:

```
public real getPeso()  
  
public void setPeso(real nuevoPeso)
```

Una propiedad simple es de sólo lectura o sólo escritura si falta uno de los mencionados métodos de acceso.

- Una **propiedad indexada** representa un conjunto de elementos, que suelen representarse mediante un vector y se identifica mediante los siguientes patrones de operaciones para leer o escribir elementos individuales del vector o el vector entero (fíjate en los corchetes del vector):

Componente
-PropiedadSimple : Tipo -PropiedadIndexada : tipo[]
+getPropiedadSimple() : Tipo +setPropiedadSimple(PropiedadSimple : Tipo) : void +getPropiedadIndexada() : Tipo [] +setPropiedadIndexada(PropiedadIndexada : Tipo []) : void +getPropiedadIndexada(posicion : int) : Tipo +setPropiedadIndexada(posicion : int, elemento : Tipo) : void

```
public <TipoProp>[] get<NombreProp>()  
  
public void set<NombreProp> (<TipoProp>[] p)  
  
public <TipoProp> get<NombreProp>(int posicion)  
  
public void set<NombreProp> (int posicion, <TipoProp> p)
```

Aquí tienes un ejemplo de propiedad indexada que resuelve el problema de Juan. El componente tiene una propiedad indexada que almacena los contenidos del menú, de este modo es sencillo crear diferentes menús de distintos tamaños. Para acceder a él solo tienes que definir dos tipos de setter y getter.


```

private String[] miembros = new String[0];

public String getMiembros(int pos){
    return miembros[pos];
}
public String[] getMiembros(){
    return miembros;
}

public void setMiembros(int pos, String miembro){
    miembros[pos] = miembro;
}
public void setMiembros(String[] miembros){
    if(miembros == null){
        miembros = new String[0];
    }
    this.miembros = miembros;
}

```

A continuación puedes descargar el código asociado para que lo estudies:

[Código para una propiedad indexada.](#) (2.27 KB)

Autoevaluación

¿Que tipo de propiedad usarías para implementar el teléfono móvil de una persona?

- ☐ Una propiedad indexada.
- ☒ Una propiedad simple.

Así es, al no tener un número variable de elementos no es preciso usar una propiedad indexada.

2.3.- Propiedades compartidas y restringidas.

Los objetos de una clase que tiene una **propiedad compartida o ligada** notifican a otros objetos oyentes interesados, cuando el valor de dicha propiedad cambia, permitiendo a estos objetos realizar alguna acción. Cuando la propiedad cambia, se crea un objeto (de una clase que hereda de **ObjetEvent**) que contiene información acerca de la propiedad (su nombre, el valor previo y el nuevo valor), y lo pasa a los otros objetos oyentes interesados en el cambio.

La notificación del cambio se realiza a través de la generación de un **PropertyChangeEvent**. Los objetos que deseen ser notificados del cambio de una propiedad limitada deberán registrarse como auditores. Así, el componente de software que esté implementando la propiedad limitada suministrará métodos de esta forma:

```
public void addPropertyChangeListener (PropertyChangeListener I)
```

```
public void removePropertyChangeListener (PropertyChangeListener I)
```

Los métodos precedentes del registro de auditores no identifican propiedades limitadas específicas. Para registrar auditores en el **PropertyChangeEvent** de una propiedad específica, se deben proporcionar los métodos siguientes:

```
public void addPropertyNameListener (PropertyChangeListener I)
```

```
public void removePropertyNameListener (PropertyChangeListener I)
```

En los métodos precedentes, **PropertyName** se sustituye por el nombre de la propiedad limitada. Los objetos que implementan la interfaz **PropertyChangeListener** deben implementar el método **propertyChange()**. Este método lo invoca el componente de software para todos sus auditores registrados, con el fin de informarles de un cambio de una propiedad.

Una **propiedad restringida** es similar a una propiedad ligada salvo que los objetos oyentes que se les notifica el cambio del valor de la propiedad tienen la opción de vetar cualquier cambio en el valor de dicha propiedad.

Los métodos que se utilizan con propiedades simples e indexadas que veíamos anteriormente se aplican también a las propiedades restringidas. Además, se ofrecen los siguientes métodos de registro de eventos:

```
public void addPropertyVetoableListener (VetoableChangeListener I)

public void removePropertyVetoableListener (VetoableChangeListener I)

public void addPropertyNameListener (VetoableChangeListener I)

public void removePropertyNameListener (VetoableChangeListener I)
```

Los objetos que implementa la interfaz **VetoableChangeListener** deben implementar el método **vetoableChange()**. Este método lo invoca el componente de software para todos sus auditores registrados con el fin de informarles del cambio en una propiedad.

Todo objeto que no apruebe el cambio en una propiedad puede arrojar una **PropertyVetoException** dentro del método **vetoableChange()** para informar al componente cuya propiedad restringida hubiera cambiado de que el cambio no se ha aprobado.

3.- Eventos. Asociación de acciones a eventos.

Juan tiene razón, la funcionalidad de un componente viene definida por las acciones que puede realizar definidas en sus métodos y no solo eso, también se puede programar un componente para que reaccione ante determinadas acciones del usuario, como un clic del ratón o la pulsación de una tecla del teclado. Cuando estas acciones se producen, se genera un **evento** que el componente puede capturar y procesar ejecutando alguna función. Pero no solo eso, un componente puede también lanzar un evento cuando sea necesario y que su tratamiento se realice en otro objeto.

Los eventos que lanza un componente, se reconocen en las herramientas de desarrollo y se pueden usar para programar la realización de acciones.

Para que el componente pueda reconocer el evento y responder ante el tendrás que hacer lo siguiente:

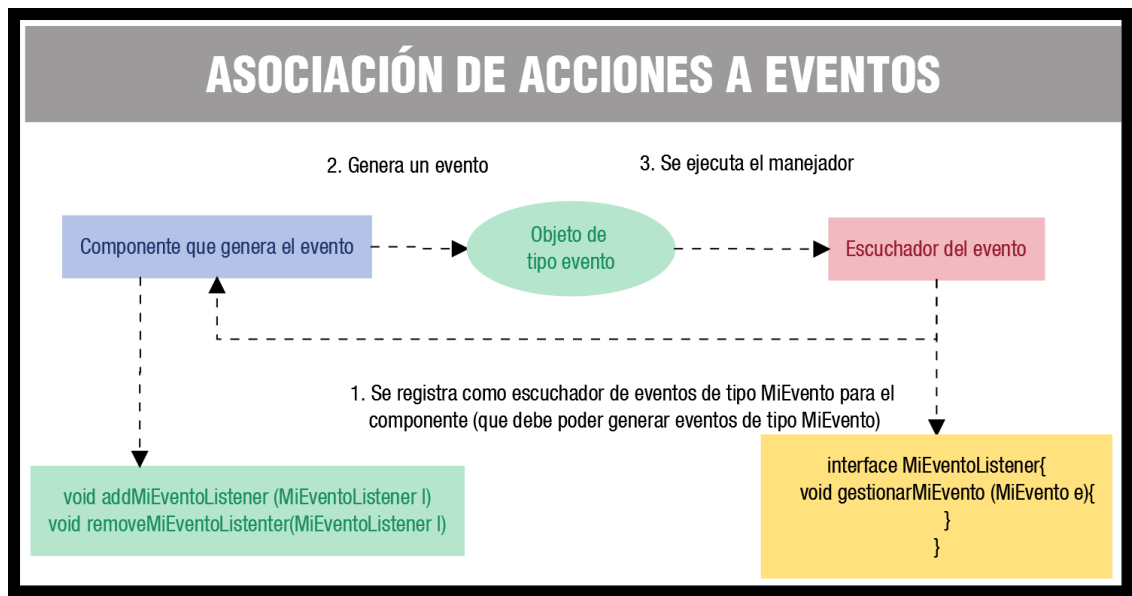
- Crear una clase para los eventos que se lancen.
- Definir una interfaz que represente el oyente (**listener**) asociado al evento. Debe incluir una operación para el procesamiento del evento.
- Definir dos operaciones, para añadir y eliminar oyentes. Si queremos tener más de un oyente para el evento tendremos que almacenar internamente estos oyentes en una estructura de datos como **ArrayList** o **LinkedList**:

```
public void add<Nombre>Listener(<Nombre>Listener I)

public void remove<Nombre>Listener(<Nombre>Listener I)
```

Finalmente, recorrer la estructura de datos interna llamando a la operación de procesamiento del evento de todos los oyentes registrados.

En resumen:



Autoevaluación

Otra manera de implementar la gestión del cambio de la propiedad `OpcionActiva` del menú de Juan podría ser...

- ☐ ...mediante una propiedad especial.
- ☐ ...programando un método especial para ello.
- ☒ ...con los eventos.

Así es, se puede generar un evento cuando se detecte que cambia esta propiedad que sea detectado en el componente panel.

3.1.- Ejemplo de gestión de eventos.

Para ver como funciona la gestión de eventos sobre componentes gráficos vamos a completar el ejemplo que estábamos viendo del campo de texto con tipo. Si recuerdas, se trataba de un componente que hereda de un campo de texto genérico.

Nuestro componente dispone, entre otras, de una propiedad llamada `texto`, que sólo puede tomar los valores "Entero", "Real" y "Texto".

El objetivo de este ejemplo es comprobar que cuando se escribe un dato éste coincida con el tipo que se indica.

Para implementar este vamos a hacer uso de un evento predefinido para los campos de texto llamado **KeyEvent**, que se dispara cuando se pulsa una tecla sobre el control. Para gestionarlo vamos a crear una función llamada **gestionaEntrada** que convertirá a nuestro componente en escuchador del evento mediante la sentencia

```
this.addKeyListener(new KeyAdapter()
```

y programará el método **KeyTyped** que gestiona que hacer con la tecla pulsada en función de si el tipo es Entero o Real, si es Texto no se gestionará la entrada porque se asume como válido cualquier texto.

Por último se llama a este método desde el constructor para que se tenga en cuenta al generar un componente nuevo.

```
@Override
public void keyTyped(KeyEvent e) {
    char character = e.getKeyChar();
    switch (tipo) {
        case "Entero":
            if (!(Character.isDigit(character)) && (character != KeyEvent.VK_BACK_SPACE))
                e.consume();
            break;
        case "Real":
            if (!(Character.isDigit(character))
                && ((character < '.') || (character > '.'))
                && (character != KeyEvent.VK_BACK_SPACE)) {
                e.consume(); // si la tecla pulsada no es un dígito, un punto o BACK_SPACE
            } else {
                if (getText().contains(".")) {
                    if (!(Character.isDigit(character))
                        && (character != KeyEvent.VK_BACK_SPACE)) {
                        e.consume(); // ignorar el evento de teclado
                    }
                }
            }
    }
}
```

Para saber más

A continuación puedes descargar el código asociado para que lo estudies:

[Código del componente con un evento implementado.](#) (4.22 KB)

4.- Introspección. Reflexión.

La **introspección** es una característica que permite a las herramientas de programación visual arrastrar y soltar un componente en la zona de diseño de una aplicación y determinar dinámicamente qué métodos de interfaz, propiedades y eventos del componente están disponibles.

Esto se puede conseguir de diferentes formas, pero en el nivel más bajo se encuentra una característica denominada reflexión, que busca aquellos métodos definidos como públicos que empiezan por get o set, es decir, se basa en el uso de patrones de diseño, o sea, en establecer reglas en la construcción de la clase de forma que mediante el uso de una nomenclatura específica se permita a la herramienta encontrar la interfaz de un componente.

También se puede hacer uso de una clase asociada de información del componente (**BeanInfo**) que describe explícitamente sus características para que puedan ser reconocidas.

Para saber más

En el siguiente enlace podrás acceder a una página web de Introducción a los JavaBeans donde podrás ampliar los conceptos de introspección, persistencia, reflexión, etc. de componentes visuales en Java.

[Características de los JavaBeans.](#)

Autoevaluación

La reflexión permite....

- ☒ ... a una herramienta de desarrollo detectar los nombres de las propiedades y métodos de la interfaz de una clase.
- ☐ ...facilitar la implementación de los métodos de una clase.
- ☐ ...conocer los elementos de la interfaz de un componente mediante el uso de una clase llamada **BeanInfo**.

Así es, mediante esta característica una herramienta de desarrollo como NetBeans puede conocer cual es la interfaz de una clase.

5.- Persistencia del componente.

A veces, necesitamos almacenar el estado de una clase para que perdure a través del tiempo. A esta característica se le llama **persistencia**. Para implementar esto, es necesario que pueda ser almacenada en un archivo y recuperado posteriormente.

El mecanismo que implementa la persistencia se llama **serialización**.

Al proceso de almacenar el estado de una clase en un archivo se le llama **serializar**. Al de recuperarlo después **deserializar**.

Todos los componentes deben persistir. Para ello, siempre desde el punto de vista Java, deben implementar los interfaces **java.io.Serializable** o **java.io.Externalizable** que te ofrecen la posibilidad de serialización automática o de programarla según necesidad:

- **Serialización Automática:** el componente implementa la interfaz **Serializable** que proporciona serialización automática mediante la utilización de las herramientas de **Java Object Serialization**. Para poder usar la interfaz serializable debemos tener en cuenta lo siguiente:
 - Las clases que implementan **Serializable** deben tener un **constructor sin argumentos** que será llamado cuando un objeto sea "reconstituido" desde un fichero .ser.
 - Todos los campos excepto **static** y **transient** son serializados. Utilizaremos el modificador **transient** para especificar los campos que no queremos serializar, y para especificar las clases que no son serializables (por ejemplo **Image** no lo es).
 - Se puede programar una **serialización propia** si es necesario implementando los siguientes métodos (las firmas deben ser exactas):
 - **private void writeObject(java.io.ObjectOutputStream out) throws IOException;**
 - **private void readObject(java.io.ObjectInputStream in) throws IOExcept**

- **Serialización programada:** el componente implementa la interfaz **Externalizable**, y sus dos métodos para guardar el componente con un formato específico. Características:
 - Precisa de la implementación de los métodos **readExternal()** y **writeExternal()**.
 - Las clases **Externalizable** también deben tener un constructor sin argumentos.

Los componentes que implementarás en esta unidad emplearán la **serialización por defecto** por lo que debes tener en cuenta lo siguiente:

- La clase debe implementar la interfaz **Serializable**.
- Es obligatorio que exista un **constructor sin argumentos**.

Para saber más

La documentación sobre la interfaz serializable de java la puedes encontrar en el siguiente enlace:

[Serialización Java.](#)

También tienes un pequeño ejemplo sobre como serializar un objeto en java.

[Como serializar un objeto en Java.](#)

6.- Otras tecnologías para la creación de componentes visuales.

En la actualidad existen diversas tecnologías para la creación de componentes visuales, además de la que hemos visto a lo largo del tema, JavaBeans implementados con NetBeans podemos encontrar otras orientaciones como los estándares COM, COM+ y DCOM de Microsoft y CORBA del Object Management Group.

En general se trata de diversas maneras de ofrecer los servicios de persistencia e introspección, para un modelo orientado a objetos de modo que se puedan crear clases reutilizables de las que se conozca su interfaz quedando oculta su implementación.

JavaBeans de Oracle

Es el estándar para crear componentes proporcionado por Oracle. Su características más destacadas las hemos visto a lo largo de la unidad, son clases Java que implementan la interfaz **Serializable** y deben disponer de un constructor sin argumentos. Define como gestionar la persistencia y la introspección y soporta propiedades simples, indexadas, compartidas o restringidas así como la gestión de eventos.

Con esta tecnología la creación de componentes visuales es realmente sencilla, basta con heredar de un control visual que ya exista, como una imagen, o una etiqueta, implementando la interfaz **Serializable** al mismo tiempo.

Como toda la tecnología Java es software libre.

El Modelo de Objetos Componentes de Microsoft.

Es la tecnología propuesta por Microsoft para la creación de componentes. Forman parte de ella COM, DCOM, COM+, OLE y ActiveX. Se basa en la creación de objetos que tiene una interfaz bien definida e independiente de la implementación de forma que pueden ser reutilizados sin más que conocer su interfaz en entornos distintos a aquel en el que fue creado. Usando DCOM además pueden estar distribuidos en varias máquinas, y COM+ usa un servidor de componentes denominado MTS.

La principal ventaja de estos componentes es que al estar separado su interfaz de su implementación se pueden usar desde diferentes lenguajes de programación, de hecho Java, Microsoft Visual C++, Microsoft Visual Basic, Delphi, PowerBuilder, y Micro Focus COBOL interactúan perfectamente con DCOM.

Aunque esta tecnología se ha implementado en muchas plataformas se usa fundamentalmente en Microsoft Windows siendo distribuido con licencia propietaria.

Para saber más

En la siguiente página web encontrarás toda la información acerca de la creación de componentes de Microsoft.

[Creación de componentes.](#)

CORBA

Es un estándar definido por el Object Management Group que permite añadir una "envoltura" a un código escrito en un determinado lenguaje con información de las capacidades que el código contiene y de sus métodos de forma que puedan ser descubiertos. Se programa en un lenguaje específico, IDL, del que existen implementaciones para diferentes lenguajes orientados a objetos, como Ada, C++, Perl, Java, SmallTalk, etc.

Para saber más

En la siguiente página web encontrarás información acerca de la creación de componentes CORBA.

[Creación de componentes CORBA.](#)

En este documento encontrarás un ejemplo de creación de una calculadora usando CORBA y Java.

[Componente calculadora CORBA.](#) (82,0 KB)

Autoevaluación

Sea cual sea la tecnología empleada para crear un componente, ¿cuál crees que es la principal característica que debe cumplir?



Que sea orientado a objetos.



Que permita la gestión de eventos.



Que permita reconocer la interfaz de un objeto con independencia de su implementación.

7.- Empaquetado de componentes.

Una vez creado el componente, es necesario empaquetarlo para poder distribuirlo y utilizarlo después. Para ello necesitarás el paquete jar que contiene todas las clases que forman el componente:

- El propio componente
- Objetos **BeanInfo**
- Objetos **Customizer**
- Clases de utilidad o recursos que requiera el componente, etc.

Puedes incluir varios componentes en un mismo archivo.

El paquete **jar** debe incluir un fichero de manifiesto (con extensión **.mf**) que describa su contenido, por ejemplo:

En este documento encontrarás un ejemplo de archivo de manifiesto:

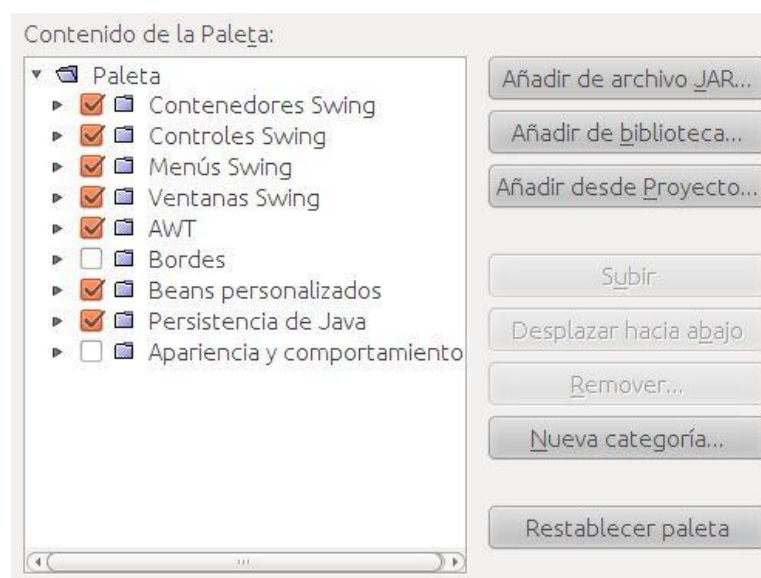
[Ejemplo de archivo de manifiesto.](#) (1.00 KB)

En el fichero de manifiesto como la clase del componente va acompañada de **Java-Bean: True**, indicando que es un **JavaBean**.

La forma más sencilla de generar el archivo **jar** es utilizar la herramienta **Limpiar y construir** del proyecto en NetBeans que deja el fichero **.jar** en el directorio **/dist** del proyecto, aunque siempre puedes recurrir a la orden **jar** y crearlo tu directamente:

```
jar cfm Componente.jar manifest.mf Componente.class ComponenteBeanInfo.class ClaseAuxiliar.class  
Imagen.png proyecto.jar
```

Una vez que tienes un componente Java empaquetado es muy sencillo añadirlo a la paleta de componentes gráficos de NetBeans, basta con abrir el administrador de la paleta, seleccionar la categoría dónde irá el componente (normalmente en Componentes Personalizados) y seleccionar el archivo jar correspondiente.



Autoevaluación

En un archivo jar se deben incluir todos los archivos necesarios para que el componente funcione, incluidos archivos de clase, imágenes e iconos, editores de propiedades, clases Customizer y archivos BeanInfo

☒ Verdadero.

☐ Falso.

Así, en el archivo de empaquetado se deben incluir todos los ficheros de clase y auxiliares necesarios para que el componente pueda ejecutarse.

8.- Elaboración de un componente de ejemplo.

Para ilustrar el proceso de elaboración de un componente usando la herramienta NetBeans, vamos a crear uno muy sencillo pero completo, en el sentido de que crea un propiedad, emplea atributos internos y genera un evento.

Se trata de un temporizador gráfico que realiza una cuenta atrás con las siguientes características:

- El componente es una **etiqueta** que dispone de una propiedad llamada **tiempo** de tipo **int** que representa los segundos que van a transcurrir desde que se inicia hasta que la cuenta llega a cero.
- Cada segundo disminuye en uno el valor de tiempo, que visualizamos en el texto de la etiqueta.
- Para programarlo se utiliza un atributo de tipo **javax.swing.Timer**, que será el que marque cuando se cambia el valor de tiempo.
- Al finalizar la cuenta atrás se lanza un evento de finalización de cuenta que puede ser recogido por la aplicación en la que se incluya el componente.

Este componente se puede utilizar, por ejemplo, en la realización de test en los que el usuario tiene que contestar una serie de preguntas en cierto tiempo.

Para la **elaboración de un componente**, debes tener muy claros los **pasos** que debes dar.

1. Creación del componente.
2. Adición de propiedades.
3. Implementación de su comportamiento.
4. Gestión de los eventos.
5. Uso de componentes ya creados en NetBeans.

En las siguientes secciones verás cómo se realizan estos pasos con el ejemplo que se acaba de exponer.

8.1.- Creación del componente.

Comenzamos creando un proyecto NetBeans nuevo de tipo Java Application. Nos aseguramos de desmarcar las opciones Crear clase principal y Configurar como proyecto principal y ponemos de nombre al proyecto ProyectoTemporizador, por ejemplo.

Una vez creado el proyecto, le añadimos un archivo nuevo de tipo Componente JavaBeans (si no lo encuentras en la lista que sale en el menú contextual, haz clic en la opción Otros... para acceder al resto de tipos de archivos). Puedes llamar a la clase **TemporizadorBean** y al paquete en el que se ubicará Temporizador.



Para que una clase se pueda considerar un componente debe implementar la interfaz **Serializable** y, además, tener un **constructor sin argumentos** que vimos eran requisitos para la creación de componentes.

El objetivo de este componente es disponer de una etiqueta con un comportamiento específico. Puesto que ya tenemos un control con parte de la funcionalidad que nos interesa, como un método para pintar el texto con el formato adecuado ya implementado, nos serviremos de el heredando nuestra clase de **JLabel**, quedando la signatura de la clase así:

```
public class TemporizadorBean extends JLabel implements Serializable
```

obviamente, tendremos que importar el paquete **javax.swing.JLabel**.

El proyecto cuenta con una propiedad de ejemplo que puedes eliminar (su declaración y los métodos **get** y **set** de la propiedad), así como un gestor de escuchadores de cambio de propiedades que no necesitaremos, quedando el siguiente código para empezar:

```
public class TemporizadorBean extends JLabel implements Serializable {  
    private PropertyChangeSupport propertySupport;  
  
    public TemporizadorBean() {  
        propertySupport = new PropertyChangeSupport(this);  
    }  
}
```

Autoevaluación

Cuando queremos crear un componente que use características de otro, como por ejemplo una etiqueta, como en el ejemplo acudimos a...

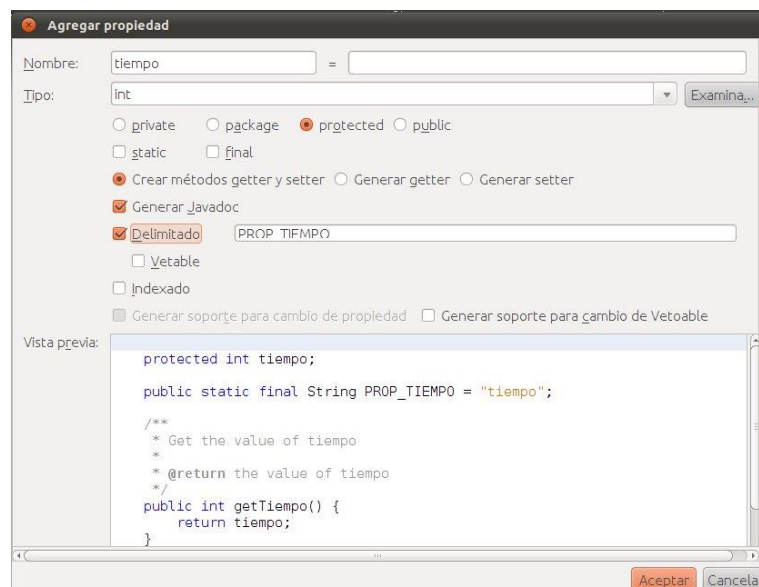
- ☐ La implementación de la clase.
- ☐ La herencia.
- ☐ El uso de una propiedad del tipo de la clase cuyas características queremos usar.
- ☐ No es posible hacer crear un componente con características de otro.

8.2.- Añadir propiedades.

El temporizador dispone de una **propiedad de lectura y escritura** en la que se almacena el número de segundos que inicialmente va a durar el temporizador y que irá disminuyendo paulatinamente cada segundo, hasta llegar a cero, momento en el que se lanzará un evento.

La adición de propiedades en una clase Java se realiza simplemente escribiendo el código de declaración del atributo privado (o protegido) y los métodos **getter** y **setter** que son la base de la introspección.

Si bien NetBeans proporciona una ayuda especial para realizar esta tarea. Con el cursor posicionado dentro de la clase (puedes reservar una zona para el código de propiedades) haz clic con el botón secundario y selecciona la opción **Insertar Código...**, en la lista que te saldrá elige **Agregar propiedad**. Se desplegará el siguiente cuadro de diálogo en el que puedes escribir el nombre de la propiedad y su tipo, e insertar los métodos **get** y **set** de manera automática:



Fíjate que si quisieras crear una propiedad indexada o restringida es muy sencillo de hacer, sólo hay que marcar la opción correspondiente y la herramienta genera el código base de los métodos necesarios.

Crea el propiedad tiempo, de tipo **int**. Aunque tenemos un código añadido, será necesario modificar algo el método **setTiempo**, ya que cada vez que cambiemos el valor de tiempo será necesario reflejar ese cambio en el texto de la etiqueta y repintarla para que surta efecto, quedando su código así:

```
public void setTiempo(int tiempo) {  
    this.tiempo = tiempo;  
    setText(Integer.toString(tiempo));  
    repaint();  
}
```

Autoevaluación

Para que una clase pueda ser considerada un componente debe implementar la interfaz **Serializable** y tener un constructor sin argumentos. ¿Verdadero o falso?



Verdadero.



Falso.

8.3.- Implementar el comportamiento.

Con la propiedad lista, tenemos que programar el comportamiento del componente. Parte de un valor inicial que irá disminuyendo cada segundo hasta llegar a cero, para implementar esto usaremos un **Timer**, que añadiremos como atributo privado de la clase. Un **Timer** se inicializa con un valor de intervalo entre acción y acción que se especifica en milisegundos, nosotros lo inicializamos a 1000 para que ejecute el método **actionPerformed** cada segundo. También hay que pasarle el objeto **actionPerformed** con el método a ejecutar, en nuestro caso, como necesitamos acceder a atributos de la etiqueta y a la propiedad tiempo, usaremos el de la propia clase. Para que la clase pueda tener el método **actionPerformed** tendrá que implementar la interfaz **ActionListener**.

A continuación tienes el enlace al código de la clase con lo visto hasta el momento:

```
public class TemporizadorBean implements ActionListener {  
    //PropertyChangeListener para el tiempo  
    private PropertyChangeListener listener;  
    //Tiempo en segundos  
    private int tiempo = 10;  
    //Timer para ejecutar la acción cada segundo  
    private Timer timer;  
    //Constructor sin argumentos  
    public TemporizadorBean() {  
        //Inicializar el timer  
        timer = new Timer(1000, this);  
        //Inicializar el listener  
        listener = new PropertyChangeListener() {  
            public void propertyChange(PropertyChangeEvent e) {  
                //Actualizar el tiempo  
                tiempo = (Integer)e.getValue();  
                //Actualizar el texto de la etiqueta  
                setText(Integer.toString(tiempo));  
                //Repintar la etiqueta  
                repaint();  
            }  
        };  
        //Añadir el listener al timer  
        timer.addActionListener(listener);  
    }  
    //Método actionPerformed  
    public void actionPerformed(ActionEvent e) {  
        //Actualizar el tiempo  
        tiempo--;  
        //Actualizar el texto de la etiqueta  
        setText(Integer.toString(tiempo));  
        //Repintar la etiqueta  
        repaint();  
        //Actualizar el tiempo  
        tiempo = 0;  
        //Desactivar el timer  
        timer.stop();  
        //Actualizar el tiempo  
        tiempo = 0;  
        //Actualizar el texto de la etiqueta  
        setText(Integer.toString(tiempo));  
        //Repintar la etiqueta  
        repaint();  
    }  
}
```

Añadiremos el siguiente código:

[Código de la clase temporizador.](#) (2.24 KB)

8.4.- Gestión de eventos.

Los componentes Java utilizan el modelo de delegación de eventos para gestionar la comunicación entre objetos como hemos visto. Para poder implementar la gestión de eventos necesitabas varias cosas:



- Una clase que implemente los eventos. Esta clase hereda de **java.util.EventObject**.
- Una interfaz que defina los métodos a usar cuando se genere el evento. Implementa **java.util.EventListener**. En este caso la gestión del evento se hará a través del método **capturarFinCuentaAtras**.
- Dos métodos, **addEventListener** y **removeEventListener** que permitan al componente añadir oyentes y eliminarlos. En principio se deben encargar de que pueda haber varios oyentes. En nuestro caso sólo vamos a tener un oyente, pero se suele implementar para admitir a varios.
- Implementar el método que lanza el evento, asegurándonos de todos los oyentes reciban el aviso. En este caso lo que se ha hecho es lanzar el método que se creó en la interfaz que describe al oyente:

Finalmente el código de la clase componente queda como aparece en el siguiente enlace:

[Código de la clase temporizador finalizada.](#) (2.81 KB)


Autoevaluación

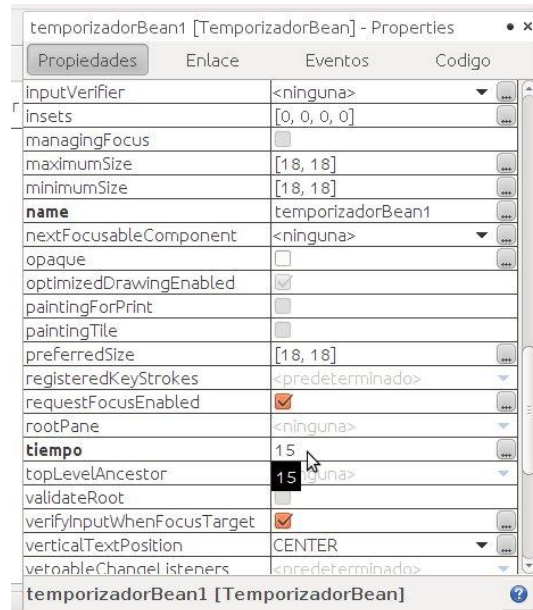
¿Qué elementos interviene en la implementación de un evento?

- ☐ Una clase que define el evento.
- ☐ Una interfaz que define los métodos a implementar.
- ☒ Una clase que defina el evento y una interfaz que defina los métodos a implementar.

8.5.- Uso de componentes previamente elaborados en NetBeans.

Una vez construido el componente es sencillo incorporarlo a la paleta de NetBeans. Podemos hacerlo de diferentes formas:

- Si lo hemos desarrollado nosotros mismos (con NetBeans) basta con utilizar Limpiar y Construir para generar el fichero **.jar** con el componente. Cuando esté generado desde el inspector de Proyectos basta con buscarlo y con el botón  seleccionar *Herramientas >> Añadir a la Paleta*. Se indica en que sección debe aparecer y ya lo tenemos.
- Si es un componente generado por otras personas, es preciso disponer de la distribución en un fichero **.jar**. E incorporarlo a la paleta desde el *Administrador de la paleta* (se accede con el botón secundario sobre la paleta). También en este caso se indica la sección en la que debe aparecer.



Una vez que hayas hecho esto tendrás el componente en la paleta y lo podrás añadir a un proyecto nuevo como cualquier otro. Esto incluye el tratamiento de las propiedades, desde el inspector de propiedades tendrás acceso a las propiedades propias de una etiqueta, y también a la propiedad tiempo definida por ti. Si la modificas verás como cambia el texto de la etiqueta.

Prueba el uso del temporizador en un proyecto nuevo. La ventaja de usar una herramienta como NetBeans es que, gracias al proceso de introspección, es capaz de detectar las propiedades, métodos y eventos del componente, por lo que es muy fácil hacer la gestión del evento de finalización de la cuenta atrás, no tienes más que hacer clic con el botón secundario sobre el componente y seleccionar *eventos >> FinCuentaAtras >> CapturarFinCuentaAtras*.

Se creará automáticamente una función que implementará la gestión del evento, puedes incluir este código:

```
private void temporizadorBean1CapturarFinCuentaAtras (
    Temporizador.TemporizadorBean.FinCuentaAtrasEvent evt) {

    // TODO add your handling code here:

    JOptionPane.showMessageDialog(null, "La cuenta atrás ha terminado", "Aviso",
        JOptionPane.INFORMATION_MESSAGE);

}
```

que simplemente muestra un mensaje, pero puedes gestionar cualquier otro modo de finalizar la cuenta atrás.

Si ejecutas el proyecto verás cómo al finalizar la cuenta atrás salta el mensaje de aviso.

Para saber más

A continuación tienes el código de la clase con el temporizador y el proyecto en el que se usa.

Para que puedas estudiar y probar el código tendrás que descomprimir la carpeta en alguna carpeta de tu disco duro. Puedes hacerlo en el directorio donde se almacenan los proyectos de NetBeans. Abre los proyectos desde **Archivo >> Abrir Proyecto**. Dado que se presentan completos puedes ver el código, compilarlos y ejecutarlos sin problemas, no obstante recuerda que para usar el componente **Temporizador** en otra aplicación debes añadirlo a la **Paleta**.

Si abres el proyecto **PruebaTemp** verás el temporizador en **JFrame**. En el panel de **Propiedades** aparece la propiedad **Tiempo**, entre el resto que pertenecen a la etiqueta de la que procede.

Analiza también el código del **ProyectoTemporizador** y observa la función que cambia cada segundo el valor de la propiedad tiempo y que lanza el evento **FinCuentaAtras** cuando llega a cero.

[Código de ejemplo.](#) (2.40 MB)