

Creación de componentes visuales

1.- Concepto de componente. Características.

Un componente es una pieza de hardware o software pequeña, que tiene un comportamiento específico y dispone de una interfaz que permite que se inserte fácilmente. Por ejemplo, una etiqueta o un botón que se basan en las etiquetas o botones estándares de la paleta de NetBeans adaptados a las necesidades específicas de una aplicación y que puede ser reutilizada es un componente. Tienen la ventaja de que pueden especializarse mucho y al estar probadas de antemano sabemos que su funcionamiento es correcto por lo que facilitan la programación de aplicaciones nuevas.

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.

Los eventos son las acciones que los usuarios van a provocar sobre los componentes.

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.

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

Un componente software no deja de ser una clase que vamos a tratar de una forma un poco diferente, así que como todas las clases se compone de un conjunto de elementos de tipos básicos a los que se llama atributos, pero si los atributos cumplen ciertas características pasarán a ser propiedades. Por ejemplo,

una etiqueta es un componente las características que la definen son sus propiedades como el color o el tipo de la fuente. Además, tiene un conjunto de métodos o funciones que permiten modificar las propiedades, y para implementar la relación con otros elementos software o con el usuario le podemos definir una serie de eventos.

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**. Recuerda que los métodos get/set se utilizan para obtener los valores de una clase.

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 siguiente 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.

Establecer propiedad **personBean1's address** utilizando: Editor predeterminado

Address 1

Address 2

City

State

Zip

En la siguiente imagen aparece el editor de una propiedad de tipo `Color`.

Establecer propiedad **etiquetaBean1's colorTexto** utilizando: Selector de color

Selector de color para GTK Paleta AWT Paleta Swing Paleta de sistema

Tono: 0 Rojo: 255

Saturación: 0 Verde: 255

Valor: 255 Azul: 255

Nombre del color: #ffffff

☒ Definir como un recurso ☐ Ubicado

Clave: etiquetaBean1.colorTexto

Valor: 255, 255, 255

Alcance: ☒ Clase ☐ Aplicación

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.

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 las 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:

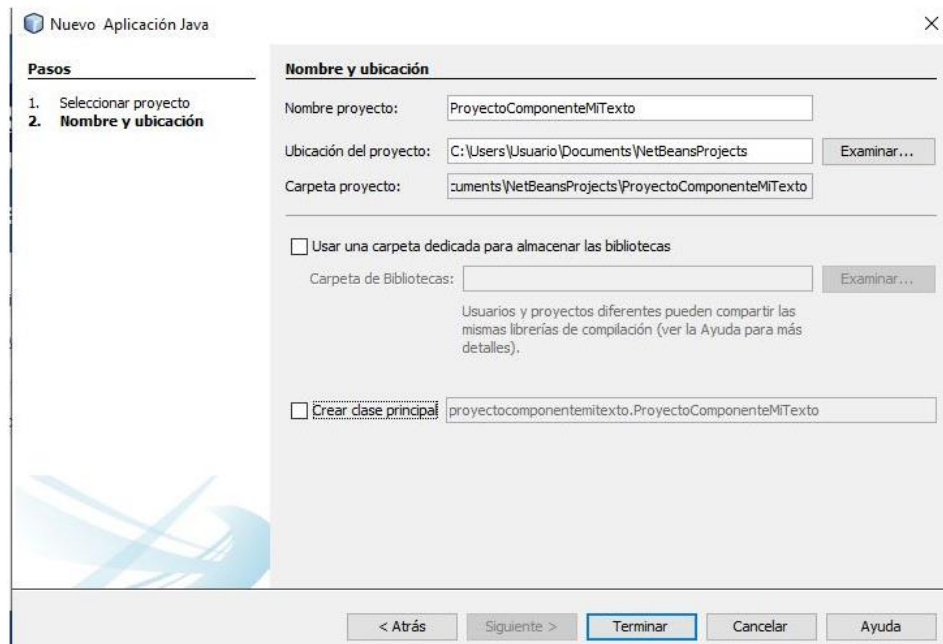
1. Crear una clase que extienda a `PropertyEditorSupport`.
2. Añadir los métodos `getAsText` y `setAsText`, que transformarán el tipo de dato de la propiedad en cadena de caracteres o viceversa.
3. Añadir el resto de métodos necesarios para la clase.
4. Asociar el editor de propiedades a la propiedad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

Para ilustrar cómo 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 cómo modificar desde NetBeans propiedades sencillas que no necesitan un editor. 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.

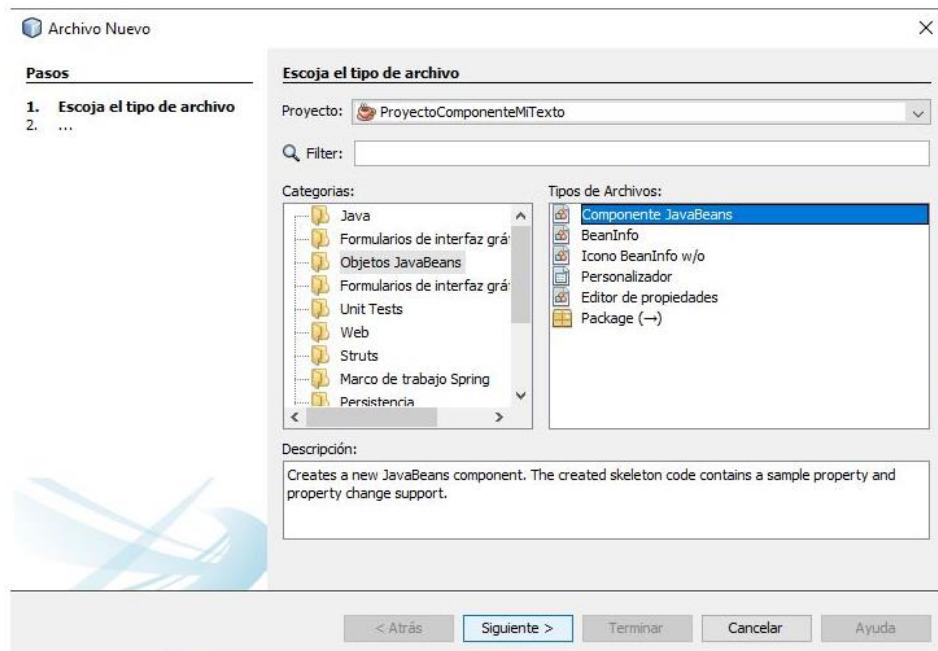
2.1.1.1.- Definición del componente.

1-. Comenzaremos por crear un proyecto de tipo Java Application. Lo configuraremos para que tenga las opciones desmarcadas de Crear clase principal y Usar una carpeta para almacenar las bibliotecas.

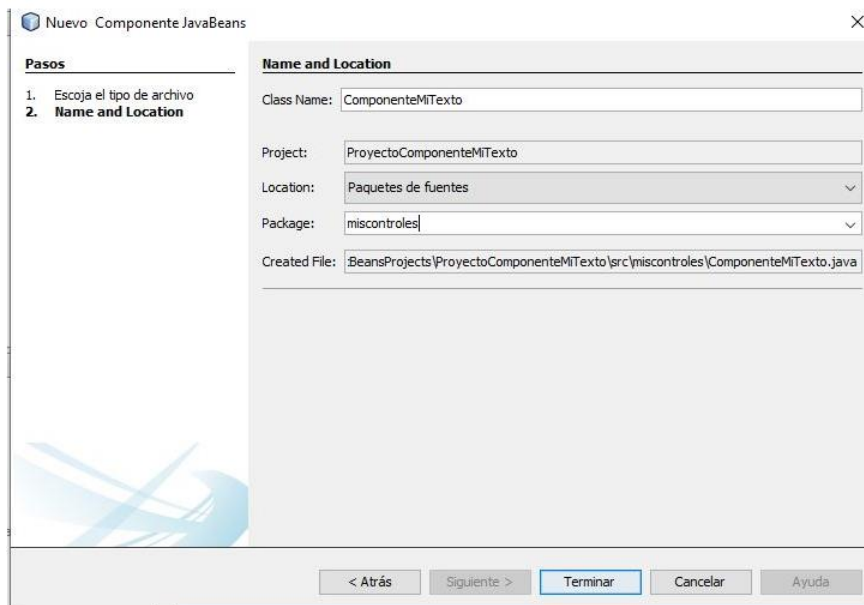


2-. A continuación, añadiremos al proyecto un archivo de tipo Componente JavaBeans.

Para ello, seleccionamos nuestro proyecto, pulsamos el botón derecho Nuevo, categorías Objeto JavaBeans y seleccionamos en la derecha el tipo de archivo mencionado.



Pulsamos *Siguiente* y le asignamos el nombre ComponenteMiTexto y lo vamos a guardar dentro de un paquete denominado miscontroles.

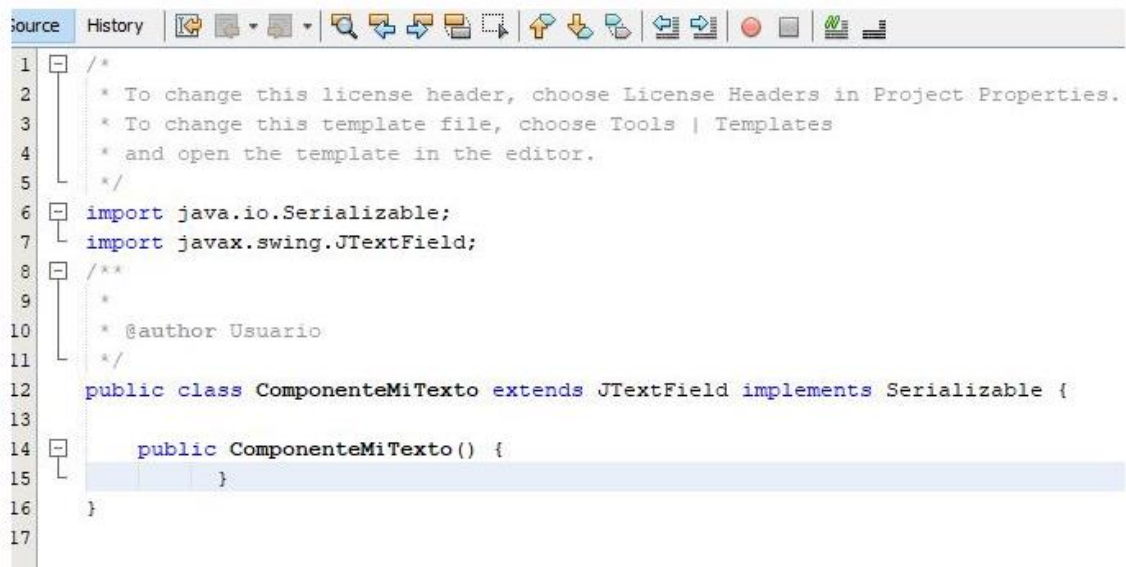


Para concluir pulsamos Terminar.

3-. Procederemos a acceder al código de la clase y eliminar todo el código introducido por el IDE. De tal forma, que la clase ComponenteMiTexto quede vacía. Sólo con la estructura básica.

```
1  /*  
2  * To change this license header, choose License Headers in  
3  * To change this template file, choose Tools | Templates  
4  * and open the template in the editor.  
5  */  
6  import java.beans.*;  
7  import java.io.Serializable;  
8  /**  
9  *  
10 * @author Usuario  
11 */  
12 public class ComponenteMiTexto implements Serializable {  
13  
14     public ComponenteMiTexto() {  
15  
16     }  
17 }  
18
```

4-. Cómo lo que vamos a crear es la modificación de una etiqueta, hacemos que la clase ComponenteMiTexto herede de la clase JTextField



```
1  /*
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  import java.io.Serializable;
7  import javax.swing.JTextField;
8
9  /**
10   *
11   * @author Usuario
12   */
13  public class ComponenteMiTexto extends JTextField implements Serializable {
14
15      public ComponenteMiTexto() {
16
17      }
```

5-. A continuación, procedemos a crear las **propiedades ancho, color y fuente**. **Ancho** lo definiremos de tipo entero, **color** de tipo *java.awt.Color* y **fuente** de tipo *java.awt.Font*. Por cada propiedad que definamos tenemos que agregar los métodos *set* y *get*. Veamos cómo se realiza para la propiedad *color* y luego este proceso habría que repetirlo para las otras dos propiedades.

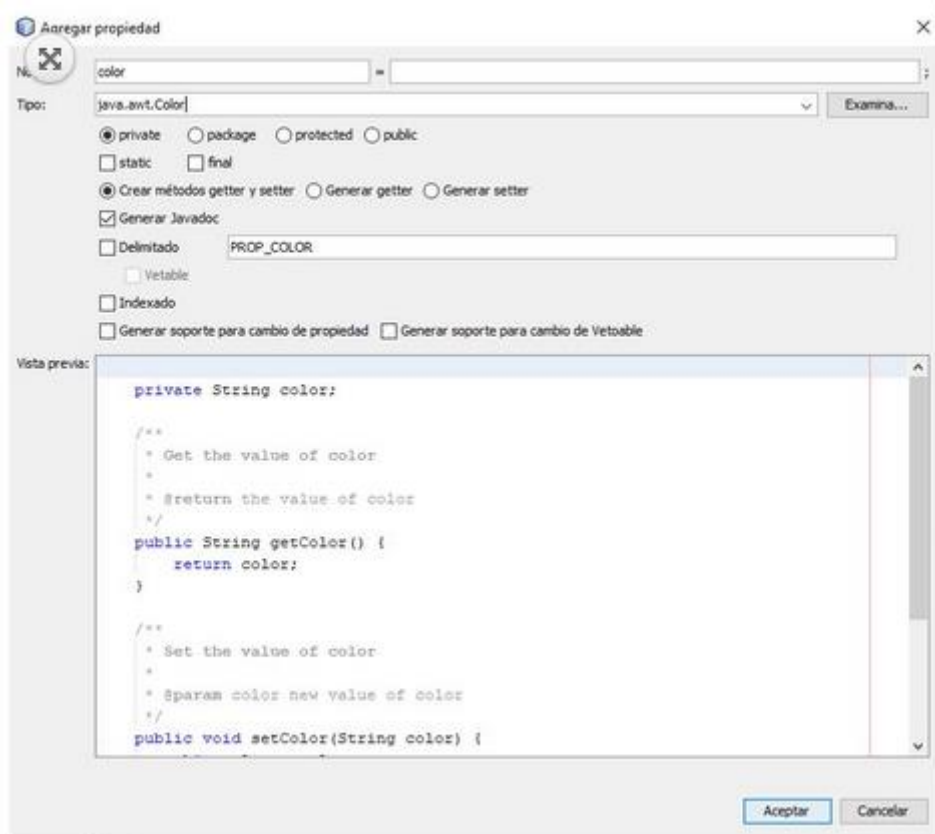


```
6  package miscontroles;
7
8  import java.beans.*;
9  import java.io.Serializable;
10 import javax.swing.JTextField;
11
12 /**
13  *
14  * @author Usuario
15  */
16 public class ComponenteMiTe
17
18
19
20 public ComponenteMiText
```

Comenzamos por desplegar el menú contextual, con el botón secundario del ratón, situándonos del código de nuestra clase y seleccionamos la opción Insertar código y a continuación, seleccionamos Agregar Propiedad.



Definimos el nombre, lo declaramos privado y activamos la casilla para que nos genere los métodos set y get.



Continuaremos por introducir el código en los métodos set y get que se han declarado. El código de la clase quedaría de la siguiente manera:


```

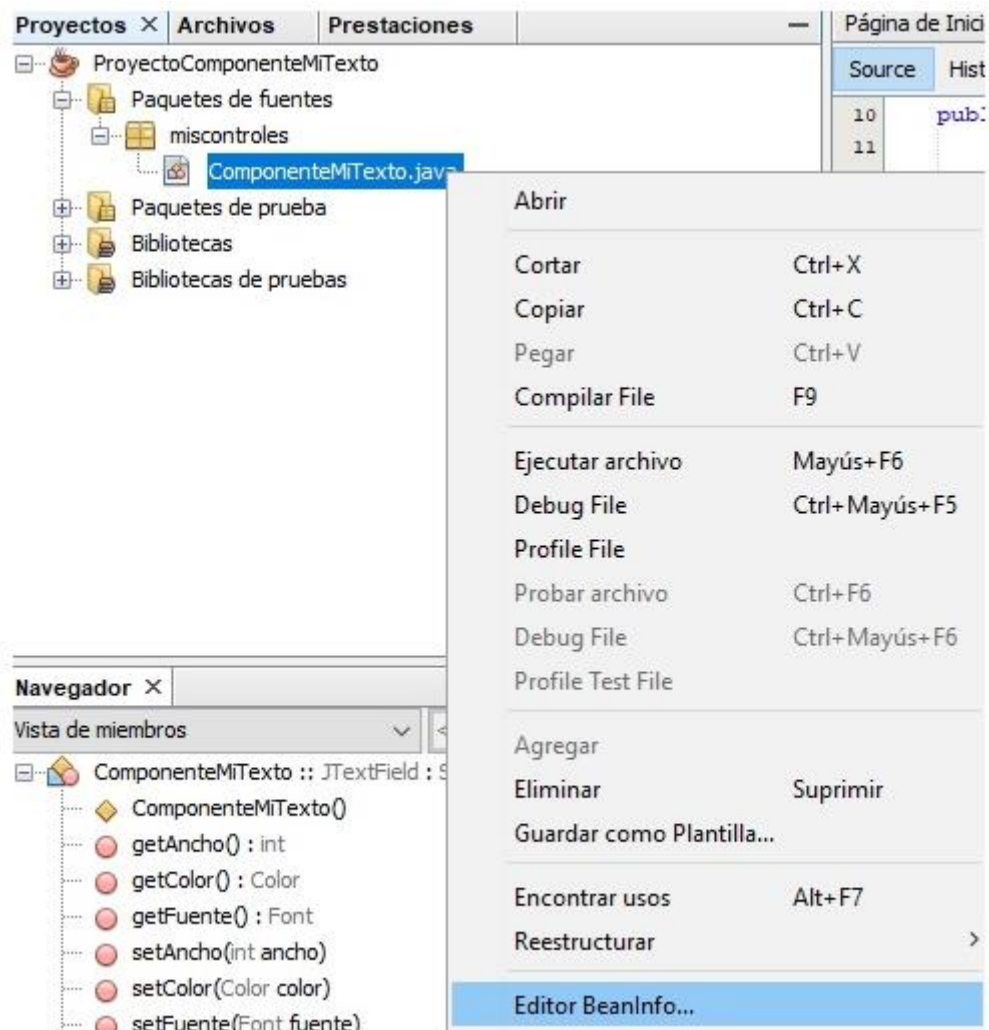
public class ComponenteMiTexto extends JTextField implements Serializable {
    private Color color;
    private int ancho;
    private Font fuente;
    public ComponenteMiTexto() {
    }
    /** Get the value of color ...5 lines */
    public Color getColor() {
        return color;
    }
    /** Set the value of color ...5 lines */
    public void setColor(Color color) {
        this.color = color;
        this.setForeground(color);
    }
    /** Get the value of ancho ...5 lines */
    public int getAncho() {
        return ancho;
    }
    /** Set the value of ancho ...5 lines */
    public void setAncho(int ancho) {
        this.ancho = ancho;
        this.setColumns(ancho);
    }
    /** Get the value of fuente ...5 lines */
    public Font getFuente() {
        return fuente;
    }
    /** Set the value of fuente ...5 lines */
    public void setFuente(Font fuente) {
        this.fuente = fuente;
        this.setFont(fuente);
    }
}

```

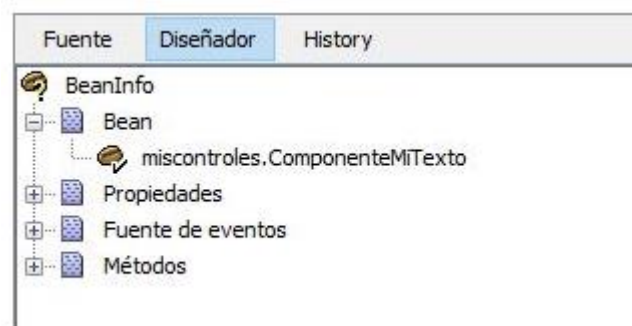
2.1.1.2.- Crear un editor de propiedades.

Una vez creado nuestro componente, el siguiente paso es asociar a las propiedades definidas un editor de propiedades. La forma más fácil es creando un objeto de tipo BeanInfo.

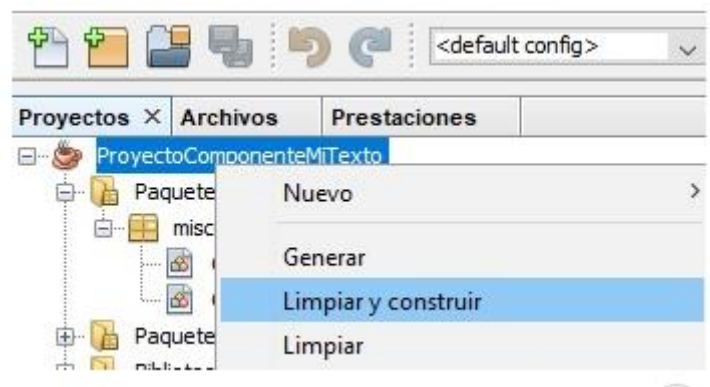
Para ello, seleccionamos la clase ComponenteMiTexto, en el panel de proyectos, y seleccionamos la opción **Editor BeanInfo**. Al seleccionarlo, nos preguntará si queremos crear un BeanInfo nuevo, responderemos que sí y veremos cómo en el panel de ficheros del proyecto nos aparece una nueva clase con el mismo nombre de la cual partimos más el texto BeanInfo. Es decir, nos habrá creado la clase ComponenteMiTextoBeanInfo.java



En la parte de la derecha, podemos ver que nos aparece el *código fuente* de la clase y el *Diseñador*. Accederemos al diseñador y podremos ver la información del nuevo JavaBean creado, podemos ver sus propiedades, métodos, etc.



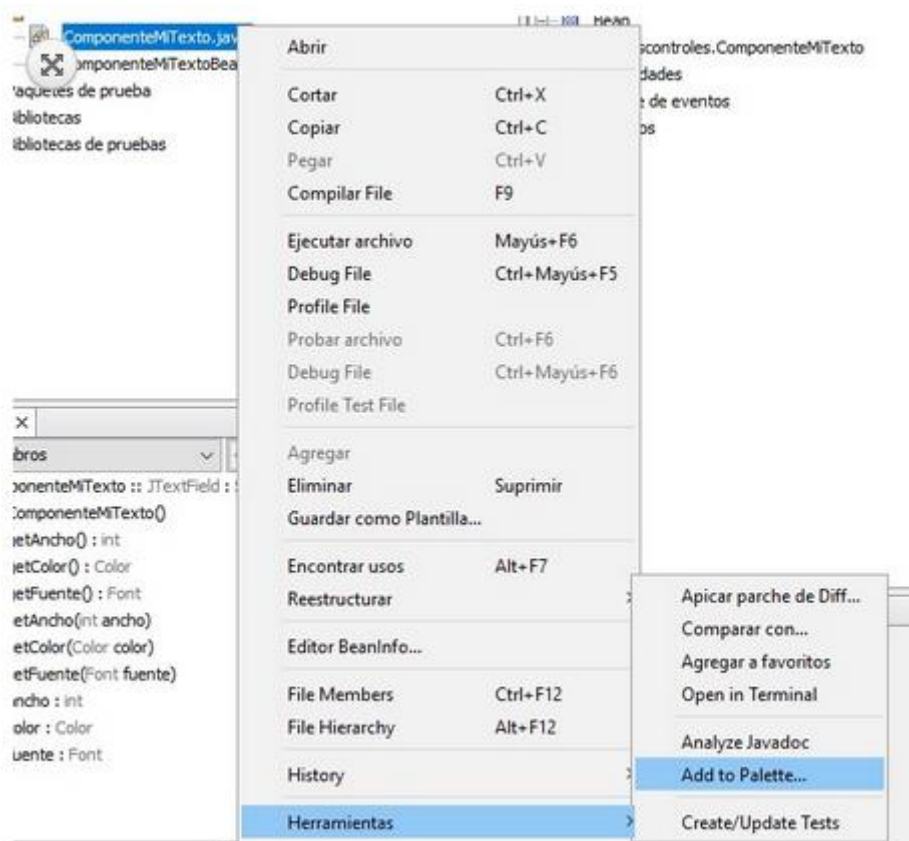
Una vez creado el JavaBean, seleccionamos nuestro proyecto, accedemos al menú contextual y seleccionamos la opción **Limpiar y Construir**. A partir de este momento ya estamos en disposición de poder utilizar nuestro componente.



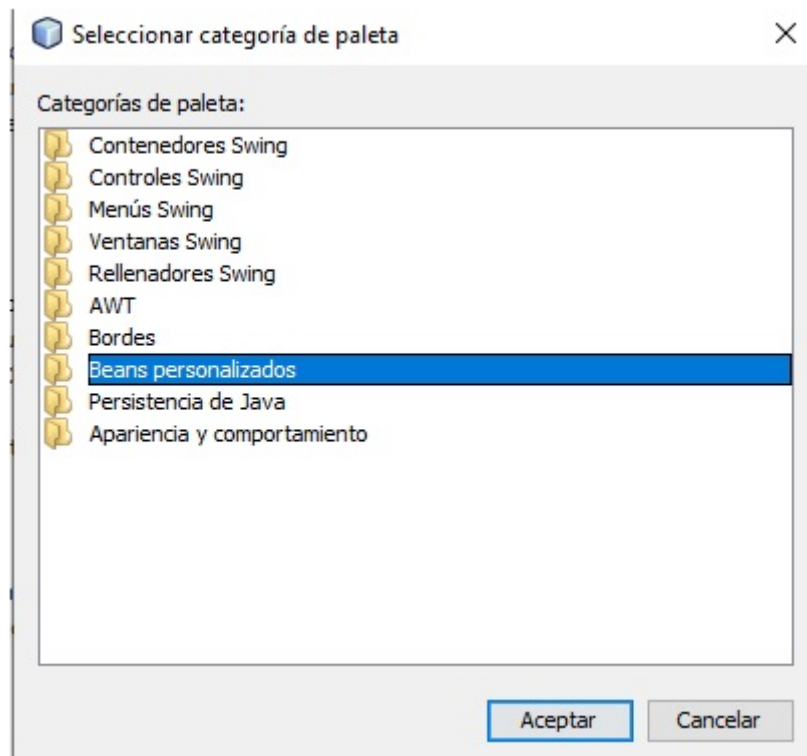
2.1.1.3.- Uso del componente.

Una vez creado y construido el componente el siguiente paso es utilizarlo. Para poder utilizarlo es necesario agregarlo a la paleta de componentes.

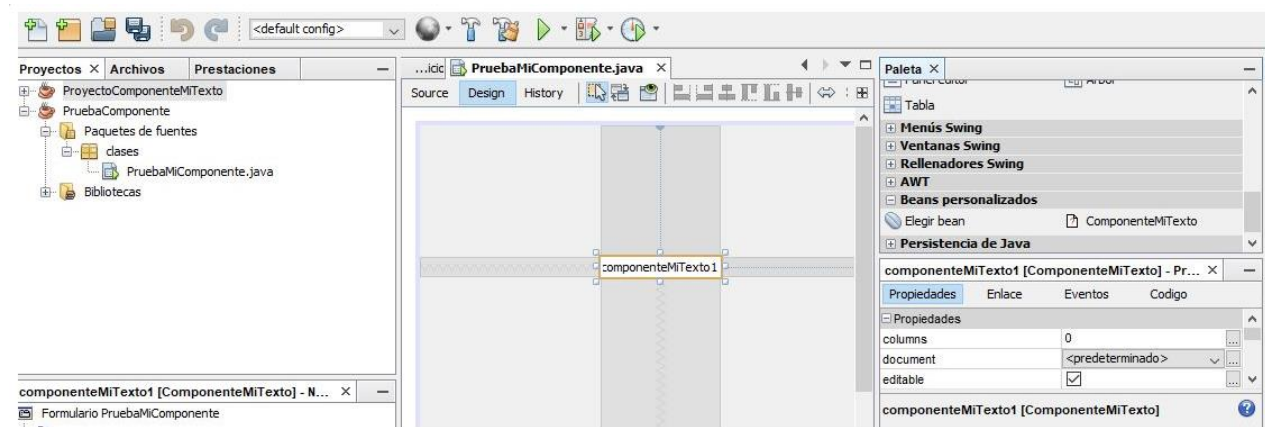
Para agregarlo a la paleta, seleccionamos la clase `ComponenteMiTexto` y accedemos al menú contextual para seleccionar la opción Herramientas (Tool) y Añadir a la paleta (Add to palette).



Al seleccionarlo, nos aparecerá una ventana para que indiquemos en qué categoría queremos introducir al componente. Se recomienda que los JavaBeans que creamos se almacenen dentro de la categoría Beans Personalizados.



Para poder probar nuestro componente, debemos de crear un nuevo proyecto de tipo aplicación Java. Le deberemos de añadir un JFrame y a continuación podemos acceder a la paleta, seleccionar nuestro componente y agregarlo al diseñador.



Una vez introducido el componente podemos acceder a sus propiedades y comprobar que podemos utilizar las características ancho, fuente y color.

2.2.- Propiedades simples e indexadas.

- Una **propiedad simple** representa un único valor, un número, verdadero o falso o un texto, por ejemplo. Por lo tanto, tomará valores dentro de un tipo de datos.

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):

```

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. 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;
}

```

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 **ObjectEvent**) 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 l)

public void removePropertyChangeListener (PropertyChangeListener l)

```

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 l)

public void removePropertyNameListener (PropertyChangeListener l)
```

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 l)

public void removePropertyVetoableListener (VetoableChangeListener l)

public void addPropertyNameListener (VetoableChangeListener l)

public void removePropertyNameListener (VetoableChangeListener l)
```

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.

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.

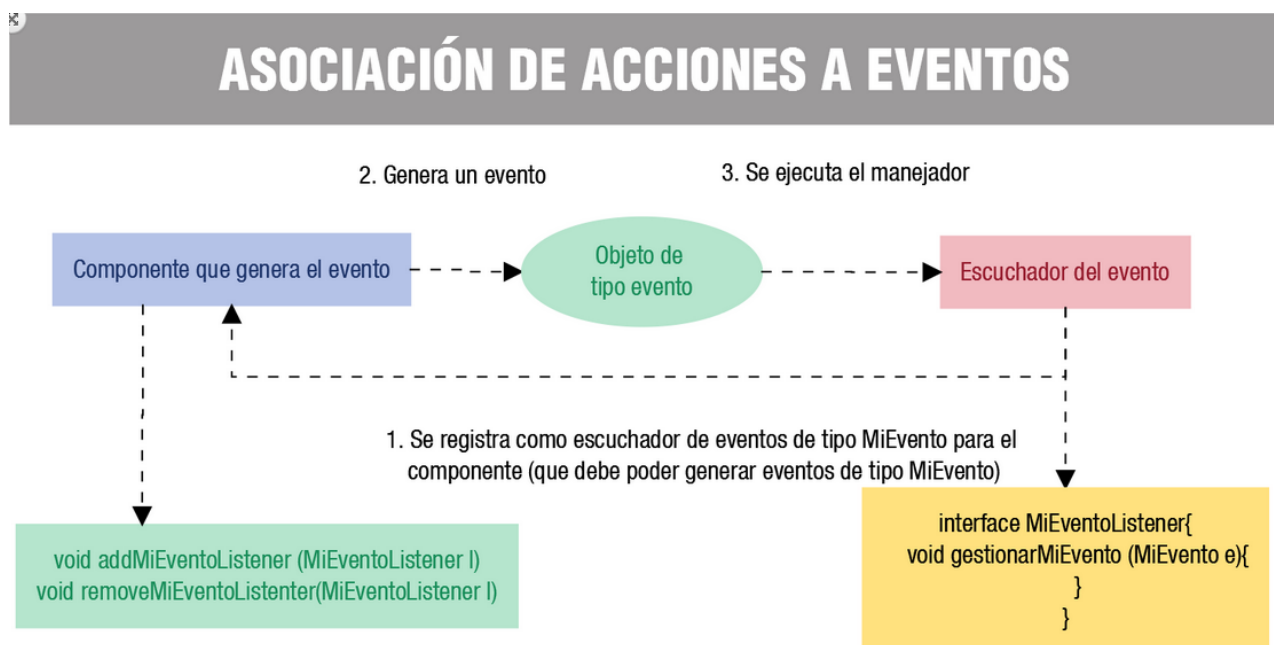
- Definir una interfaz que represente el oyente (listener) asociado al evento. Debe incluir una operación para el procesamiento del evento.
- Definir dos operaciones, para añadir y eliminar oyentes. Si queremos tener más de un oyente para el evento tendremos que almacenar internamente estos oyentes en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

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

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

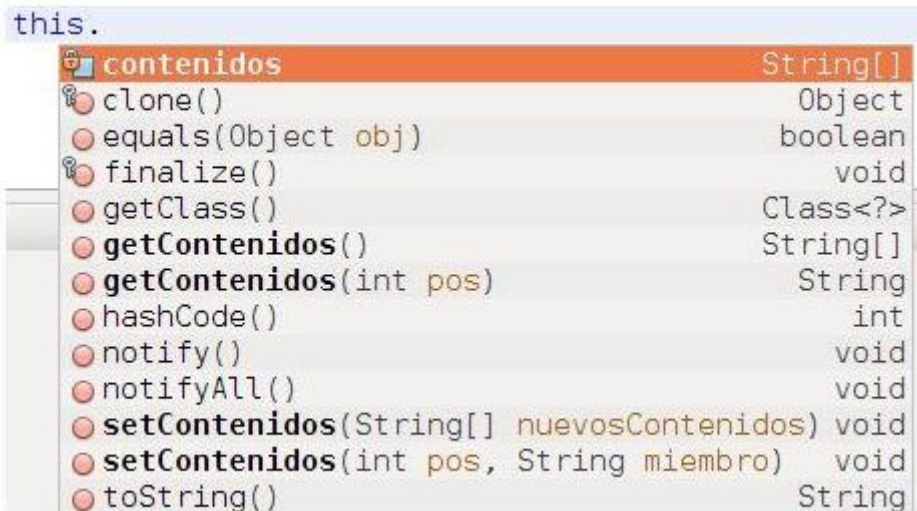
- Finalmente, recorrer la estructura de datos interna llamando a la operación de procesamiento del evento de todos los oyentes registrados.

En resumen:



4.- Introspección. Reflexión.

Cuando trabajamos con cualquier clase, y los componentes no dejan de serlo, el editor proporciona algunas ayudas a la edición, te dan pistas acerca de los métodos que puedes usar o qué argumentos debes colocar en una función. ¿Cómo puede la herramienta conocer eso?, sencillamente porque un componente, como cualquier otra clase dispone de una interfaz, que es el conjunto de métodos y propiedades accesibles desde el entorno de programación. Normalmente, la interfaz la forman los atributos y métodos públicos.



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.

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.

<http://www.sc.ehu.es/sbweb/fisica/cursoJava/applets/javaBeans/fundamento.htm>

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.
 - Se puede programar una **serialización propia** si es necesario implementando los siguientes métodos (las definiciones de los métodos deben ser exactas):
 - *private void writeObject(java.io.ObjectOutputStream out) throws IOException;*
 - *private void readObject(java.io.ObjectInputStream in) throws IOException;*
- **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**.

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.

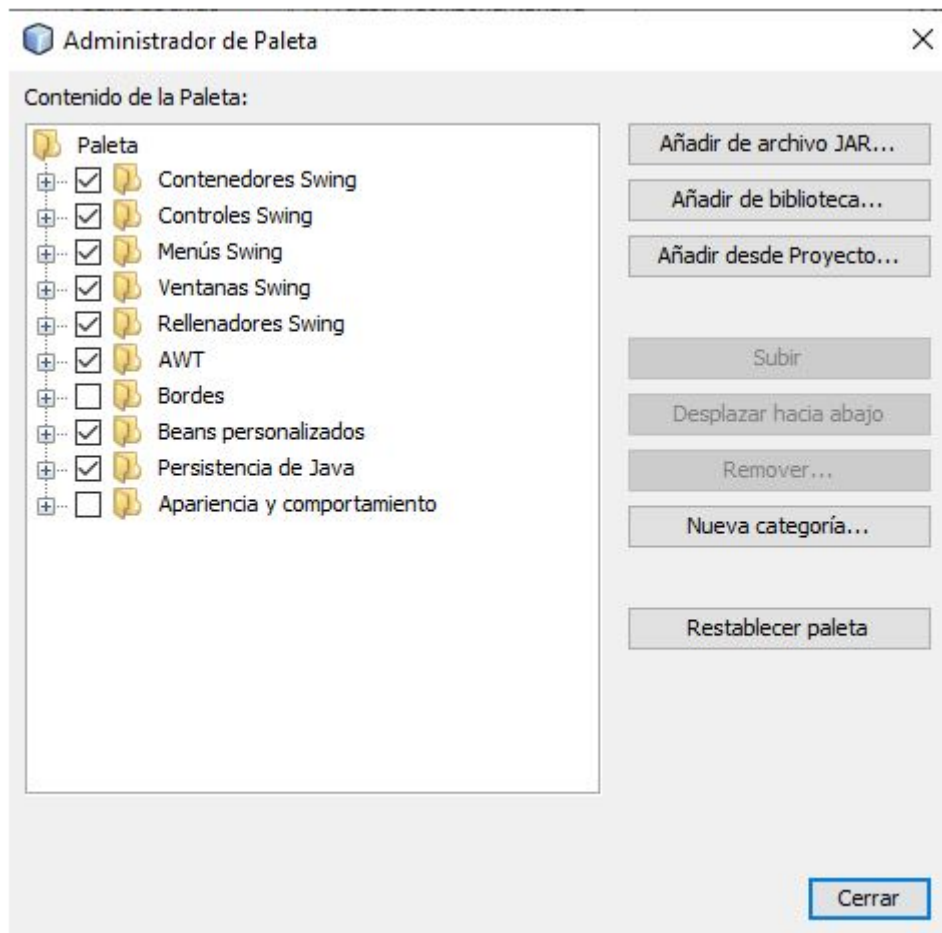
Puedes incluir varios componentes en un mismo archivo.

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

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. Para acceder al administrador de la paleta, nos situamos encima de la paleta (panel en donde se encuentran definidos los componentes), pulsamos el botón derecho del ratón (menú contextual) y elegimos la opción Administrador de paleta.



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 una 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.

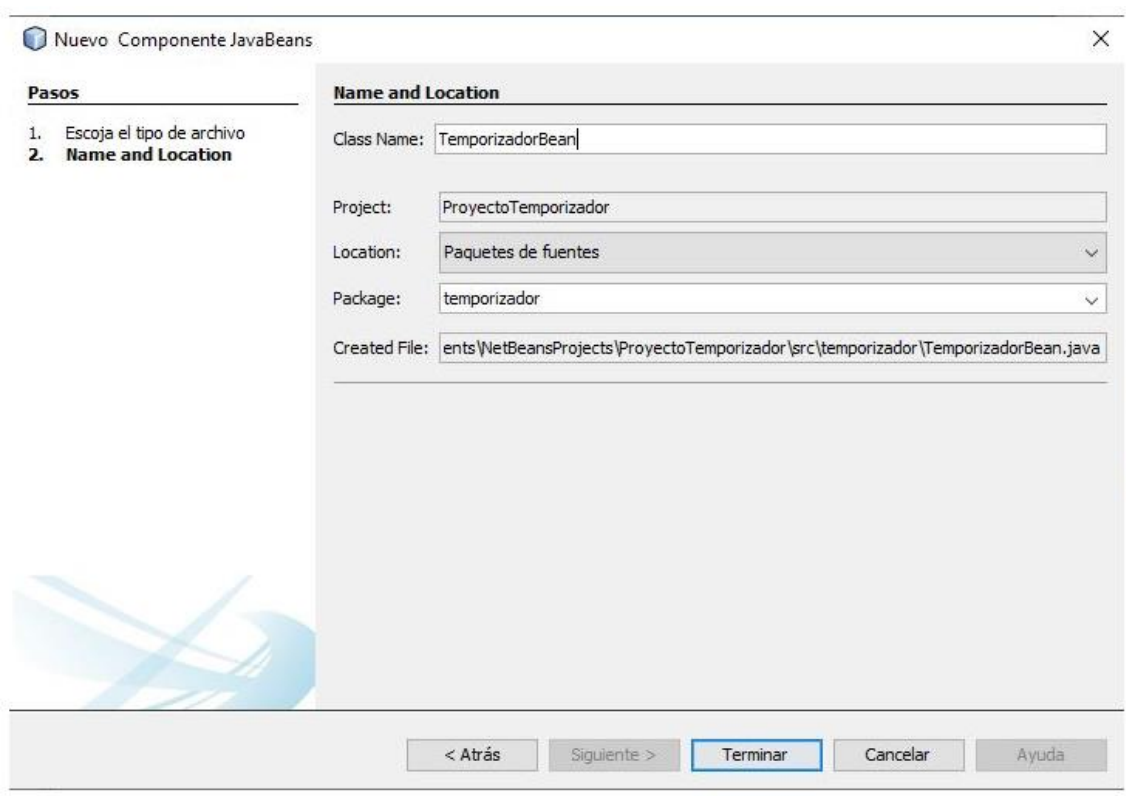
1. Creación del componente.
2. Adición de propiedades.
3. Implementación de su comportamiento.
4. Gestión de los eventos.
5. Uso de componentes ya creados en NetBeans.

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

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. Le podemos añadir un paquete con el nombre ***Temporizador***.

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 se ubicará en el paquete ***Temporizador***.



El proyecto:

Procederemos a acceder al código de la clase que 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. De tal forma, que la clase `TemporizadorBean` quede vacía. Sólo con la estructura básica.

```
package Temporizador;

import java.io.Serializable;

/**
 *
 * @author Angel
 */
public class TemporizadorBeans implements Serializable {

    public TemporizadorBeans() {

    }

}
```

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 él heredando nuestra clase de `JLabel`, quedando la signatura de la clase así:

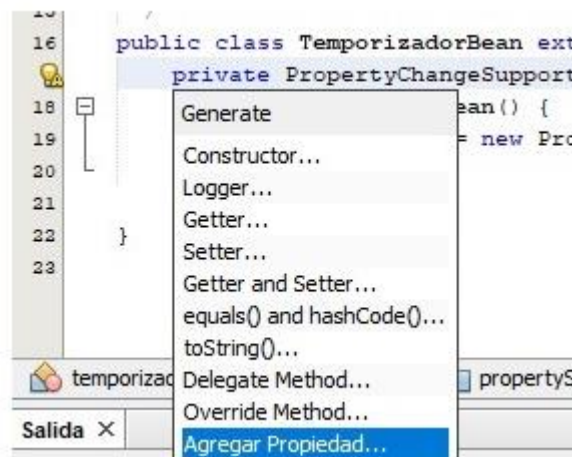
```
public class TemporizadorBeans extends JLabel implements Serializable {
```

Al realizar esto se ha de insertar de forma automática `import javax.swing.JLabel`; Si no se realiza así habrá que realizarlo de forma manual.

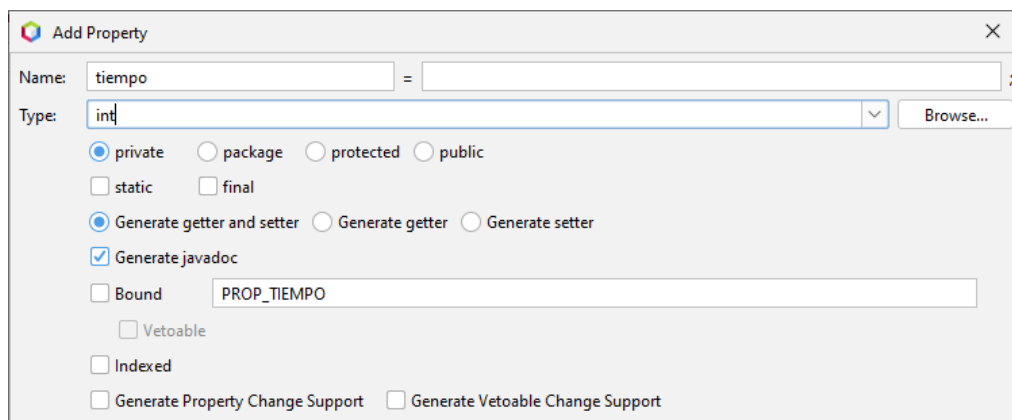
Añadir propiedades.

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 la propiedad tiempo, de tipo **int**. Esto hará que se añada el siguiente código.

```
/**
 * Get the value of tiempo
 *
 * @return the value of tiempo
 */
public int getTiempo() {
    return tiempo;
}

/**
 * Set the value of tiempo
 *
 * @param tiempo new value of tiempo
 */
public void setTiempo(int tiempo) {
    this.tiempo = tiempo;
}
```

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();
}
```

Implementar el comportamiento.

Una vez definida la propiedad y su comportamiento, 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**.

Como acabamos de indicar al constructor **TemporizadorBean** creado sin argumentos, le vamos a añadir el código necesario para definir un objeto **Timer** programado cada segundo, es decir, cada 1000 milisegundos. En el constructor, inicializamos nuestro contador con el valor 5. Por lo tanto, la cuenta atrás comenzará en dicho valor. Además, activamos temporizador.

El método constructor sin argumentos quedaría definido de la siguiente forma:


```
public TemporizadorBean() {
    tiempo = 5; //Para que comience la cuenta atrás a los 5 segundos
    t = new Timer (1000, this); //Inicialización del objeto Timer
    setText(Integer.toString(tiempo)); //Método que modifica el texto a
visualizar por la etiqueta
}
```

Ahora, además, debemos importar la librería Timer.

```
import javax.swing.Timer;
```

Definir la variable t.

```
private final Timer t;
```

Y modificar la signatura de la clase añadiendo ActionListener, quedando así:

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

Esto debería importar de forma automática las librerías.

```
import java.awt.event.ActionListener;
import java.awt.event.ActionEvent;
```

Y haber añadido el método ***actionPerformed***,

```
@Override

public void actionPerformed(ActionEvent e) {

    throw new UnsupportedOperationException("Not supported yet."); //
Generated from
nbfs://nbhost/SystemFileSystem/Templates/Classes/Code/GeneratedMethodBody

}
```

A nuestra clase le añadiremos los métodos:

public final setActivo(boolean valor): contendrá el código necesario para gestionar si nuestro temporizador, objeto Timer está funcionando o no.

```
//Activo es en sí mismo una propiedad (sin atributo subyacente)
//Gestiona si el temporizador está funcionado o no.
public final void setActivo(boolean valor) {
    if (valor == true)
        t.start();
    else
        t.stop();
}
```

public boolean getActivo(): contendrá el código necesario para que nos devuelva si nuestro temporizador se está ejecutando o no.

```
public boolean getActivo() {
    return t.isRunning();
}
```

```
}
```

En el constructor añadimos el código `setActivo(true)` ; para activar el temporizador.

```
public TemporizadorBeans() {  
    tiempo = 5; //Para que comience la cuenta atrás a los 5 segundos  
    t = new Timer (1000, this); //Inicialización del objeto Timer  
    setText(Integer.toString(tiempo)); //Método que modifica el texto a visualizar por la etiqueta  
    setActivo(true);  
}
```

A continuación, modificaremos el método ***actionPerformed***, para que cada vez que sea invocado se disminuya nuestro contador en una unidad:

```
public void actionPerformed(ActionEvent e){  
    setText(Integer.toString(tiempo)); //Asignamos el valor a mostrar en la  
    etiqueta  
    repaint(); //Repintamos la etiqueta  
    tiempo--; //Disminuimos el valor de la variable tiempo para que vaya  
    disminuyendo hasta 0.  
    if(tiempo== 0){  
        //Cuando alcancemos el valor cero mostramos un aviso de Terminado.  
        setActivo(false);  
        JOptionPane.showMessageDialog(null, "Terminado", "Aviso",  
        JOptionPane.INFORMATION_MESSAGE);  
    }  
}
```

Será necesario importar la siguiente librería para poder utilizar los mensajes de dialogo.

```
import javax.swing.JOptionPane;
```

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:

1-. Crearemos una clase que implemente los eventos. Esta clase hereda de ***java.util.EventObject***.

Para continuar con nuestro ejemplo, vamos a añadir a nuestro proyecto, una nueva clase con el nombre ***FinCuentaAtrasEvent*** que herede de la clase ***java.util.EventObject***

```
public class FinCuentaAtrasEvent extends EventObject{  
  
}
```

Que importará la siguiente librería

```
import java.util.EventObject;
```

Modificamos el código de esta clase, con lo que quedaría de la siguiente manera:

```
public class FinCuentaAtrasEvent extends EventObject{
    public FinCuentaAtrasEvent(Object source) {
        super(source);
    }
}
```

2.- Debemos de **definir una interfaz** que defina los métodos a usar cuando se genere el evento. Implementa ***java.util.EventListener***.

Para continuar con nuestro ejemplo, procederemos a añadir dentro de la clase *TemporizadorBean.java* una nueva interfaz con el nombre *FinCuentaAtrasEvent*. En este caso la gestión del evento se hará a través del método ***capturarFinCuentaAtras***.

El código necesario para la definición de la interface sería el siguiente:

```
package temporizador;
public interface FinCuentaAtrasListener extends EventListener{
    public void capturarFinCuentaAtras (FinCuentaAtrasEvent ev);
}
```

Será necesario importar la siguiente librería.

```
import java.util.EventListener;
```

3-. Añadiremos dos métodos, ***addEventoListener*** y ***removeEventoListener*** 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. Estos métodos deben de añadirse dentro de la clase que estemos utilizando para definir el componente.

Siguiendo con nuestro ejemplo, accedemos a la clase ***TemporizadorBean.java***. Añadiremos los métodos ***addFinCuentaAtrasListener*** y ***removeFinCuentaAtrasListener***. El código que debemos de añadir es el siguiente:

```
public void addFinCuentaAtrasListener(FinCuentaAtrasListener receptor){
    this.receptor = receptor;
}
public void removeFinCuentaAtrasListener(FinCuentaAtrasListener receptor){
    this.receptor=null;
}
```

Debemos definir la variable *receptor*.

```
private FinCuentaAtrasListener receptor;
```

4-. 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.

En el ejemplo, modificaremos el método *actionPerformed* de la clase *TemporizadorBean.java* para que cuando el temporizador llegue a cero, lance el evento. El código de este método una vez que lo modifiquemos quedaría de la siguiente manera:

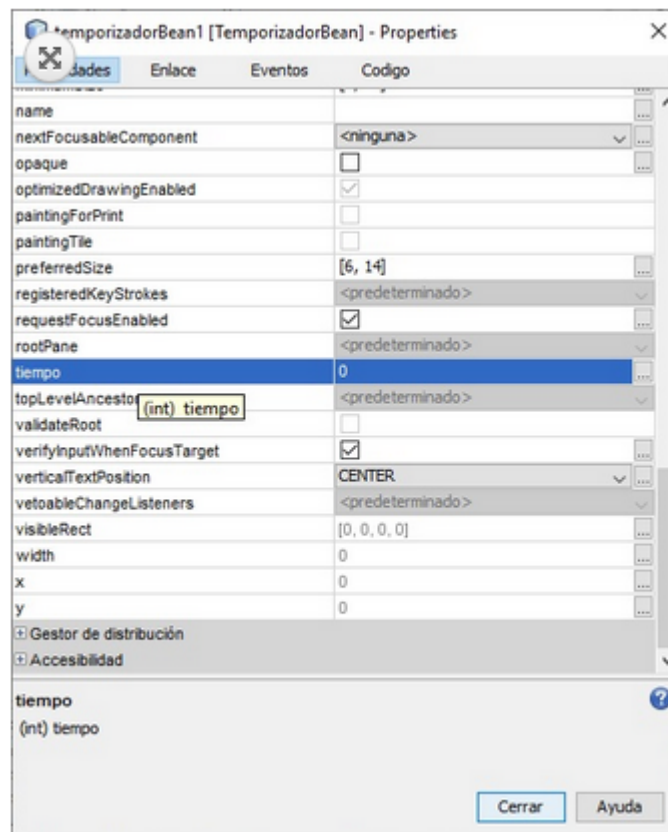
```
public void actionPerformed(ActionEvent e) {
    setText(Integer.toString(tiempo));
    repaint();
    tiempo--;
    if(tiempo == 0){
        setActivo(false);
        receptor.capturarFinCuentaAtras(new FinCuentaAtrasEvent(this));
    }
}
```

Uso de componentes previamente elaborados en NetBeans.

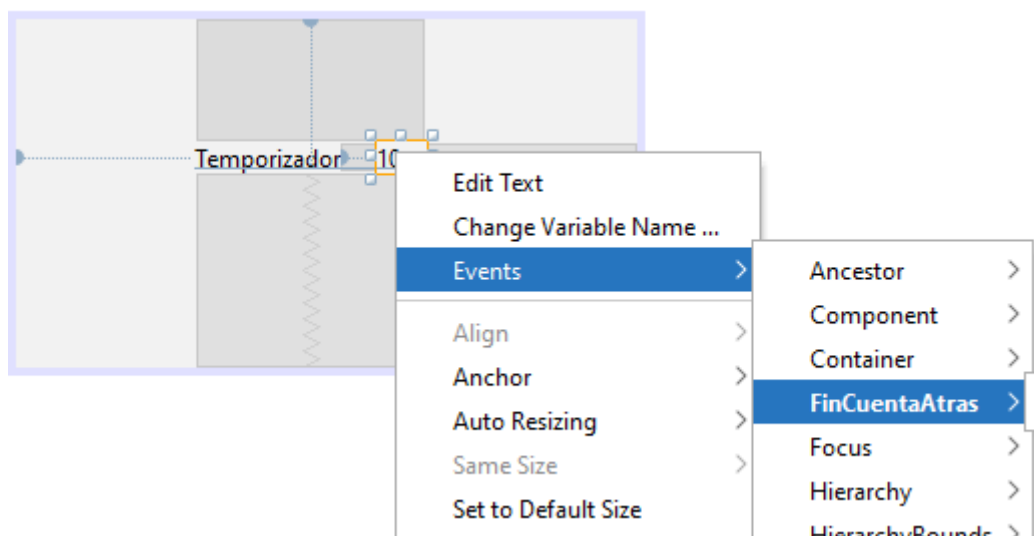
Una vez construido el componente es sencillo incorporarlo a la paleta de NetBeans. Podemos hacerlo de diferentes formas:

- 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 secundario 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 cómo 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 cómo 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.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.

