.

TEMA 3

CREACIÓN DE COMPONENTES VISUALES

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**[1] 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.

Reflexiona

Un componente es una unidad de composición de software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto de otros componentes, de forma independiente en tiempo y en espacio.

Szypersky, 1998

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**[2], es decir, debe poder guardar el estado de sus propiedades cuando han sido modificadas.

[1] Suceso que ocurre en un sistema, como puede ser un clic, minimizar una ventana, etc.

[2] Característica que permite almacenar el estado de una clase para que perdure a través del tiempo.

- Debe tener **introspección**[3], 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, gracias principalmente a la rápida reutilización de código que implica.
- Se disminuyen los errores en el *software* ya que los componentes se deben someter a un riguroso control de calidad antes de ser utilizados (además, la sucesión de versiones acaba por producir un componente cada vez más afinado).

Reflexiona

Ya has creado algunas interfaces gráficas con diferentes herramientas como NetBeans o Designer de QT. En ambos casos, el procedimiento consiste en seleccionar los controles gráficos de la interfaz y posicionarlos en un lienzo que será la interfaz. Todos estos controles cumplen con los requisitos para ser componentes, de hecho lo son, son elementos reutilizables que pueden ser manejados por una herramienta de desarrollo visual. Por ejemplo, se puede modificar su tamaño o color para adaptarlo a la interfaz y estos cambios permanecen después de cerrarla, además tienen una interfaz formada por un conjunto de métodos y propiedades accesibles desde la paleta de propiedades y la capacidad de responder a eventos, por ejemplo el clic del ratón.

[3] Propiedad de un componente del diseño de interfaces, mediante la cual es posible visualizar el interior del mismo; esto es, sus propiedades, métodos y eventos disponibles.

Elementos de un componente: propiedades y atributos

Como en cualquier clase, un componente tendrá definido un estado a partir de un conjunto de **atributos**. Un atributo es una variable cuyo estado viene determinado por su nombre y su **tipo de datos**[4] (los cuales toman valores concretos). Normalmente los atributos son privados, lo que implica que no son visibles desde fuera de la clase que implementa el componente, usándose 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 las cuales acceden a dicha propiedad, pudiendo ser de dos tipos:

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

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

Si la propiedad es **booleana**[5] 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)
```

[4] Atributo de los datos que maneja un ordenador y que le confiere determinadas propiedades: qué valores puede tomar, qué operaciones se pueden realizar, etc.

[5] Tipo de dato que sólo admite un valor de dos posibles: verdadero o falso.

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) // Establece el color del botón
public String getColor() // Obtiene el color del botón
```

Reflexiona

En el lenguaje de programación Java los componentes se crean utilizando la tecnología **JavaBeans**, que consiste en crear una clase con unas características especiales que puede ser reutilizada después de una manera muy sencilla. De hecho es común que a un componente Java se le llame **Bean**. A continuación, tienes un enlace a un ejemplo de creación de un Bean.

Introducción a los JavaBeans

Creación de un JavaBean (tutorial)

JavaBeans (otro tutorial)

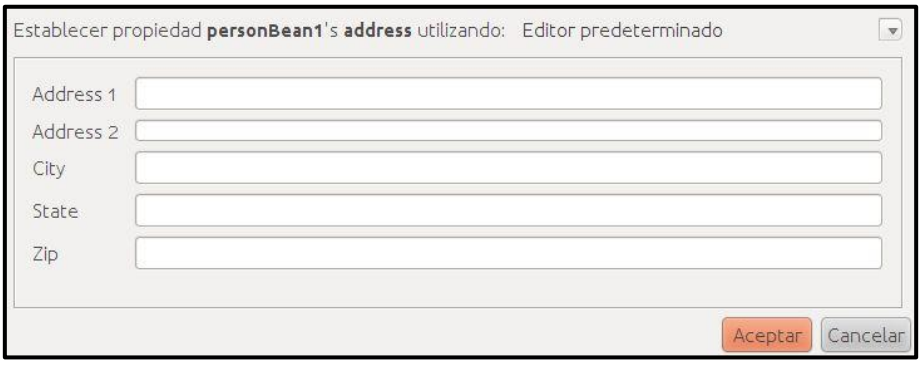
La principal función de un componente es recoger en una entidad que sea fácilmente reutilizable un segmento de código con una cierta funcionalidad cerrada. Aporta ventajas tales como la reutilización, disminución de costes, mejora de la robustez e incremento de la tolerancia a fallos de los nuevos programas ya que suelen estar ampliamente probados.

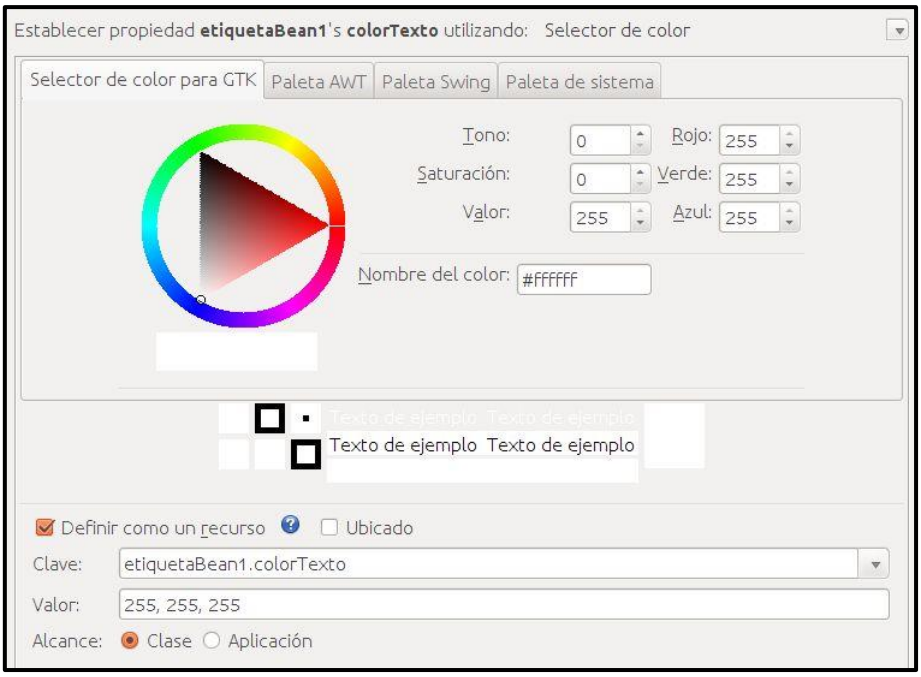
Modificar gráficamente el valor de una propiedad con un editor

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

Ejemplo de creación de un componente con un editor de propiedades

Reflexiona

Un JavaBean es un componente software reusable que puede ser manipulado visualmente mediante una herramienta gráfica.

Especificación de JavaBeans, Sun 1997

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.

Imagen una ventana de aplicación Java, con la imagen de una taza de café humeante.

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 **fuente**. De esta manera veremos cómo 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 **fuente** 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 sí que necesitará que programemos un editor de propiedades para él.

Veamos cómo 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:

Proyecto de NetBeans para descargar

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**[6] 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 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 de una propiedad indexada

[6] Tipo de dato compuesto formado por un conjunto de datos, todos del mismo tipo, posicionados de forma unívoca.

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**[7]. 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.*

[7] Conjunto de objetos que implementan la interfaz **PropertyChangeListener**. Son objetos que están a la escucha de los cambios que se producen en determinadas propiedades de un componente, y que si implementan el método **propertyChange()** pueden hacer que se ejecute una acción ante el cambio de la 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.

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**[8] que represente el **oyente**[9] (**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**:

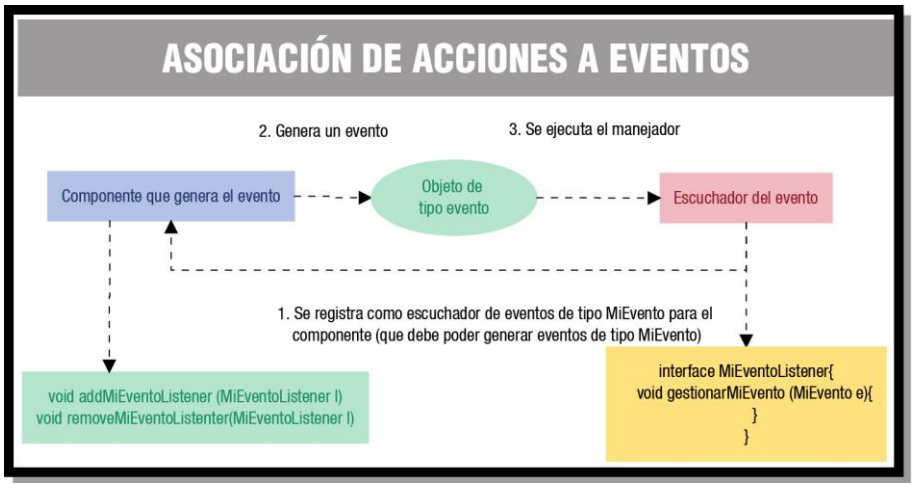
```
public void add<Nombre>Listener(<Nombre>Listener l)
public void remove<Nombre>Listener(<Nombre>Listener l)
```

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

En resumen:

[8] Programa que actúa como intermediario entre el ordenador y el usuario, formado por un conjunto de imágenes y objetos gráficos que representan la información y acciones disponibles.

[9] Sinónimo de listener y escuchador de eventos. Es una sentencia que escribimos directamente en el código y nos permite saber cuándo ocurre un evento.



Ejemplo de gestión de eventos

Para ver cómo 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 **KeyTiped** que gestiona qué 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 caracter = e.getKeyChar();
    switch (tipo) {
        case "Entero":
            if (!(Character.isDigit(caracter)) && (caracter != KeyEvent.VK_BACK_SPACE))
                e.consume();
            }
            break;
        case "Real":
            if ((!Character.isDigit(caracter))
                    && ((caracter < '.') || (caracter > '.'))
                    && (caracter != KeyEvent.VK_BACK_SPACE)) {
                e.consume();  // si la tecla pulsada no es un digito, un punto o BACK_SI
            } else {
                if (getText().contains(".")) {
                    if ((!Character.isDigit(caracter))
                            && (caracter != 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

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**[10], 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

Desde los siguientes enlaces podrás acceder a distintos sitios web con información sobre JavaBeans, en los que podrás ampliar los conceptos de introspección, persistencia, reflexión, etc. de componentes visuales en Java.

Introducción a los JavaBeans

Acceso a datos. Introducción a los JavaBeans

Una introducción a JavaBeans

[10] Mecanismo para visualizar las características de un componente.

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** y, 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 artículo sobre los secretos de la serialización con algunos ejemplos

Serialización de objetos en Java

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 los 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 estructura, creación y uso de componentes de Microsoft.

El modelo de componentes COM

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:

Programación de objetos distribuidos con CORBA

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

Componente calculadora CORBA

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

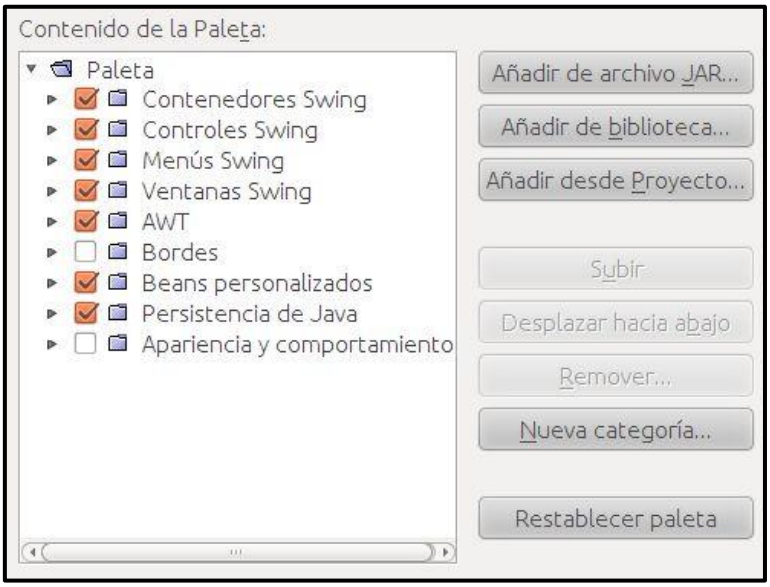
https://www.youtube.com/watch?v=1X0PmX0l_34

*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 tú directamente mediante:

```
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.



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.

Como es difícil programar un componente desde cero, porque hay que programar desde cómo se renderiza en pantalla, como responde a los eventos … son muchas líneas de código, lo que vamos hacer es a partir de la base de uno de estos y a través del mecanismo de herencias extenderlo y crear nuestro componente a partir de uno de estos.

Vamos a crear un panel a partir de la clase JPanel de java y poner una imagen de fondo , luego poner componentes encima con la imagen de fondo creada.

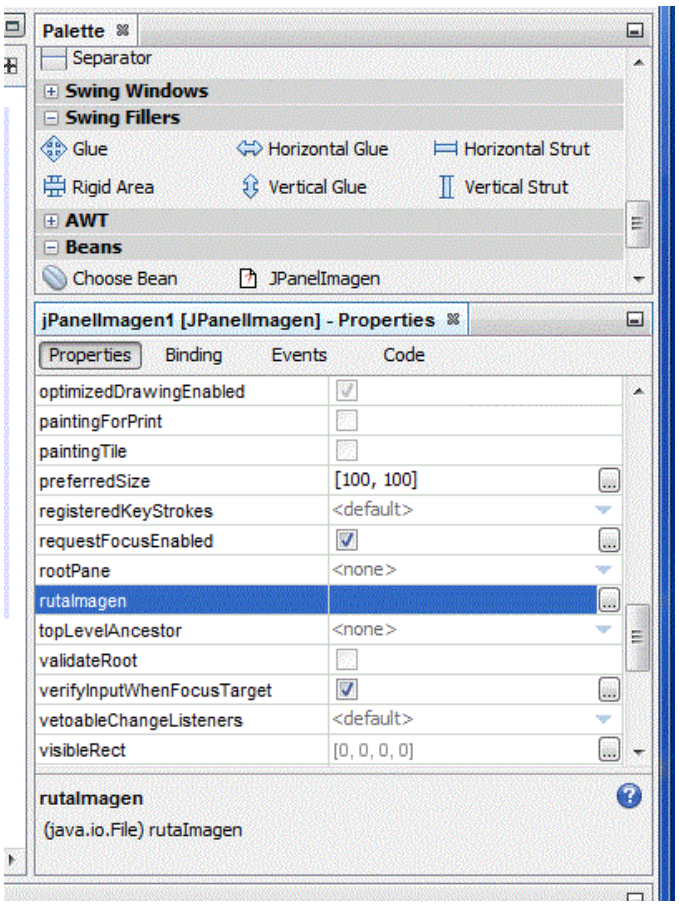
En el siguiente videos se muestra :

1. Creación del componente
2. Añadir las propiedades
3. Implementar
4. Gestionar los eventos.
5. Uso del componente previamente elaborado en Netbeans.

https://www.youtube.com/watch?v=oIaTAS6-G4k
https://www.youtube.com/watch?v=c1F9J4Lu_LI

Uso de componentes previamente elaborados en NetBeans

Una vez construido el componente es sencillo incorporarlo a la palea de Netbeans y poder utilizar en otros proyectos java. En el video anterior vimos como realizar este proceso.



ÍNDICE