



Centro Integrado de Formación Profesional
AVILÉS
Principado de Asturias

UNIDAD 3: DISEÑO DE COMPONENTES

DESARROLLO DE INTERFACES

2º CURSO

C.F.G.S. DESARROLLO DE APLICACIONES MULTIPLATAFORMA

REGISTRO DE CAMBIOS

Versión	Fecha	Estado	Resumen de cambios
1.0	21/10/2023	Aprobado	Primera versión completa
1.1	26/10/2023	Aprobado	Añadidas referencias a ejemplos
1.2	04/11/2023	Aprobado	Añadido apartado correspondiente a invocación a servicio RESTFUL
1.3.	14/11/2023	Aprobado	Modificaciones en apartado de implementación.
1.4.	27/11/2023	Aprobado	Añadido apartado de uso de JComboBox

ÍNDICE

REGISTRO DE CAMBIOS	1
ÍNDICE	1
UNIDAD 3: DISEÑO DE COMPONENTES	2
3.1. Concepto de componente. Características	2
3.1.1. Validación de datos	3
3.1.2. Aspecto de los componentes	5
3.1.3. Uso del componente JComboBox	5
3.2. Herramientas para el desarrollo de componentes visuales	9
3.3. Propiedades, atributos y métodos.	10
3.3.1. Editores de propiedades	12
3.3.2. Propiedades simples e indexadas	25
3.4. Eventos, asociación de acciones a eventos	27
3.5. Introspección. Reflexión	29
3.5.1. Reflexión	29
3.6. Persistencia del componente	34
3.7. Prueba de los componentes	36
3.8. Empaquetado de componentes	37
3.9. Caso práctico: Conexión de componente a servicio REST	39
ÍNDICE DE FIGURAS	52
BIBLIOGRAFÍA – WEBGRAFÍA	52

UNIDAD 3: DISEÑO DE COMPONENTES

3.1. 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 se sabe 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. 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.

El primer requisito que debe cumplirse para crear un nuevo componente es que este sea un **JavaBean**, lo que implica que debe cumplir dos requisitos:

- Implementar la interfaz [Serializable](#)
- Tener definido un constructor por defecto

Ejemplo 1: Cuadro de texto con colores de fondo y texto:

Un primer ejemplo puede ser la creación de un componente que permita introducir un texto. El componente tendrá una propiedad de tipo `Color`, que será el color de fondo del componente cuando el usuario haya introducido un número de caracteres superior al número indicado por una segunda propiedad de tipo entero. En apartados posteriores se verá cómo desarrollarlo contextualizándolo con los aspectos vistos.

Ejemplo 2: Panel con imagen de fondo (JPanelImagen)

Un segundo ejemplo va a consistir en crear un nuevo componente que permita una imagen de fondo y seleccionar la transparencia de dicha imagen.

En primer lugar, se crea un proyecto de aplicación Java (por ejemplo, con Maven) que puede tener como nombre **JPanelImagen**. Dentro de ese proyecto, se creará una clase con el mismo nombre **JPanelImagen**:

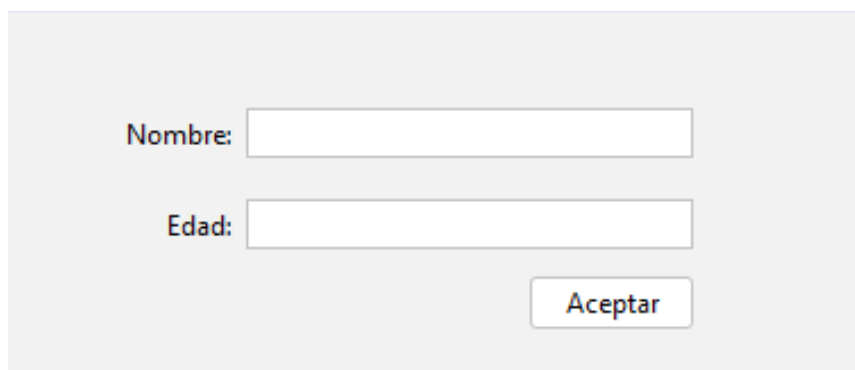
```
import java.io.Serializable;
import javax.swing.JPanel;
public class JPanelImagen extends JPanel implements Serializable
{
    public JPanelImagen(){
    }
}
```

Como puede verse, la clase hereda de **JPanel** y además cumple con los requisitos para ser un **JavaBean**: implementar **Serializable** y tener un constructor por defecto.

3.1.1. VALIDACIÓN DE DATOS

Tanto en componentes como en formularios, es necesario conocer los mecanismos de validación de los datos que se introducen por el usuario. A continuación, se verá este proceso desde un enfoque tradicional.

Como ejemplo de uso, se creará un control a partir de un panel llamado **PanelValidacion** el cual será añadido a un proyecto de ejemplo con un aspecto como el de la figura:



1. Panel para validación tradicional

Los controles pueden tener los siguientes nombres: txtNombre, txtEdad y btnAceptar. Se puede escribir un método que devuelva un booleano si los controles tienen valores válidos:

```
private boolean validarComponente(){
    String nombre = txtNombre.getText();
    if (nombre == null || "".equals(nombre)){
        JOptionPane.showMessageDialog(this, "El nombre no puede estar vacío", "Error
en nombre", JOptionPane.ERROR_MESSAGE);
        return false;
    }
    try{
        Integer.valueOf(txtEdad.getText());
    }
    catch(NumberFormatException e){
        JOptionPane.showMessageDialog(this, "El campo edad tiene que ser un número
entero", "Error en edad", JOptionPane.ERROR_MESSAGE);
        return false;
    }
    return true;
}
```

En este caso, primero se comprueba que el campo txtNombre tiene algún valor devolviendo false en caso contrario. La siguiente porción de código valida que el valor del cuadro de texto referente a la edad es un entero. Con el método valueOf de la clase Integer, se puede realizar esa conversión y, en caso de fallo, mostrar un mensaje de error (todo ello envuelto en un bloque try...catch).

Ahora se puede llamar al método desde el manejador del botón y añadir una variable que recoja el valor devuelto por este método:

```
private boolean valido = true;
private void btnAceptarActionPerformed(java.awt.event.ActionEvent evt)
{
    valido = validarComponente();
}
```

De esta forma, el componente es válido hasta que se pulse el botón. Si los campos no están validados, esta variable valdrá false. Para exponerla en el componente, se puede aprovechar la propiedad [isValid](#) que tiene la clase Component la cual devuelve true cuando el componente está bien posicionado y con tamaño correcto en su contenedor padre:

```
@Override
public boolean isValid() {
    return valido && super.isValid();
}
```

Este método sobrescrito primero comprueba si es válido por el criterio que se ha determinado en relación con sus campos y después evalúa si lo es por los que tiene el método de la clase base. El orden en este caso es importante ya que la expresión primero evaluará la variable

valido, la cual tiene una lógica muy sencilla y que, en caso de ser false, ya cortocircuitará la expresión y no seguirá evaluando más (dando por hecho que el método de la clase base tiene una lógica más compleja, lo que redundará en más recursos de computación).

3.1.2. ASPECTO DE LOS COMPONENTES

Para una aplicación Java susceptible de ejecutarse en cualquier plataforma, el aspecto de los componentes debe adaptarse a cada plataforma. Java provee una solución con el mecanismo de look and feel. Cada componente gráfico está de hecho formado por dos clases. Una gestiona el aspecto funcional del componente, mientras que la segunda afecta únicamente al aspecto visual de este. Esta última se adapta en función de la plataforma sobre la que se ejecuta la aplicación. La clase [UIManager](#) es responsable del funcionamiento de este mecanismo. Cuando se crea un componente gráfico, el constructor del componente utiliza la clase **UIManager** para obtener una instancia de la clase encargada de gestionar el aspecto gráfico del componente. La clase UIManager debe estar informada del look and feel que debe utilizar. En esta URL se pueden encontrar ejemplos de Looks And Feel:

<https://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/plaf.html#available>

Es posible indicar el look and feel que debe utilizarse de tres formas diferentes:

- Mediante programación: utilizando el método **setLookAndFeel** de la clase **UIManager** pasando como parámetro el nombre completo de la clase que implementa el look and feel deseado.

```
UIManager.setLookAndFeel("javax.swing.plaf.nimbus.NimbusLookAndFeel");
```

Es aconsejable realizar la modificación del look and feel al comienzo de la aplicación antes incluso de la creación del primer componente gráfico.

- Mediante un parámetro de la línea de comandos utilizada para ejecutar la aplicación.

```
java -Dswing.defaultlaf=javax.swing.plaf.nimbus.NimbusLookAndFeel aplicacion
```

También se puede utilizar un archivo de configuración swing.properties para definir el look and feel por defecto que aplicará la máquina virtual Java durante la ejecución de la aplicación gráfica. Este archivo debe estar situado en la carpeta lib de la máquina virtual. Debe contener la siguiente línea:

```
swing.defaultlaf=javax.swing.plaf.nimbus.NimbusLookAndFeel
```

3.1.3. USO DEL COMPONENTE JCOMBOBOX

En la unidad 1 se introdujo uno de los componentes que trabaja con una lista de elementos estructurados en forma de tabla, el JTable. JComboBox es un componente asociado al cuadro desplegable que se puede encontrar en cualquier interfaz de usuario, tanto de escritorio como web y/o en dispositivos móviles. Es interesante tener en cuenta que una buena parte de lo que se detalla a continuación se puede aplicar a otro elemento con una naturaleza similar: el elemento JList.

En primer lugar, se debe conocer de qué forma se puede llenar un JComboBox. En el diseño que ofrecen los IDE, este componente se compone de una colección de elementos (ítems) de tipo cadena (String); pero la potencia de un componente de esta naturaleza se explota a partir de poder incluir en él objetos completos, ya que lo normal es que este tipo de componentes recoja los resultados obtenidos por una consulta en una base de datos. El ejemplo más habitual es aquel en el que el desplegable muestra una lista de registros de una tabla base de un modelo. Por ejemplo, si en dicha tabla se recogen las provincias de un país, es bastante probable que conste de dos columnas: "id" y "nombre". Asociada a esa tabla, se debería tener una clase con variables miembro del mismo nombre y tipo que las de la tabla subyacente. El código siguiente podría ser un ejemplo de la clase asociada a la tabla "Provincia".

```
public class Provincia {  
    private int id;  
    private String nombre;  
    public Provincia() {  
    }  
    public Provincia(int id, String nombre) {  
        this.id = id;  
        this.nombre = nombre;  
    }  
    public int getId() {  
        return id;  
    }  
    public void setId(int id) {  
        this.id = id;  
    }  
    public String getNombre() {  
        return nombre;  
    }  
    public void setNombre(String nombre) {  
        this.nombre = nombre;  
    }  
    @Override  
    public String toString() {  
        return this.nombre;  
    }  
}
```

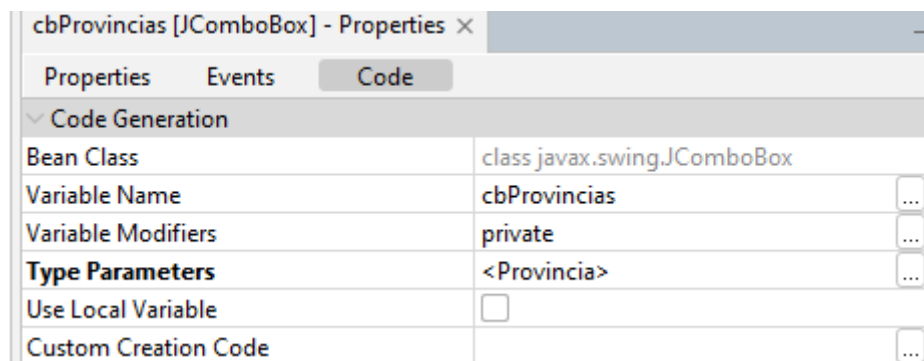
La idea es que el desplegable contenga objetos de este tipo y poder obtener el identificador de la provincia seleccionada mostrando su nombre al usuario. Por supuesto, teniendo el objeto como elemento, se pueden obtener todas aquellas propiedades que se deseen, aunque en este caso no haya más que estas dos, id y nombre. Algo que se debe hacer además, es sobrescribir el método toString() con la propiedad que se desee que se muestre. En este caso, es el nombre.

Para poder integrar este objeto, existen varias opciones. La primera se lleva a cabo de forma similar a como se hizo en su momento con la JTable, es decir, creando lo que sería un ComboBoxModel.

Aunque se podría crear una clase que herede de [DefaultComboBoxModel](#) esto solo sería necesario en caso de requerir algún cambio importante en la clase base. Para este caso, no se necesitaría más que instanciar un objeto de este tipo y añadirle los elementos necesarios. Por ejemplo:

```
DefaultComboBoxModel<Provincia> cbModel = new DefaultComboBoxModel<>();  
cbModel.addAll(lstProvincias);  
cbProvincias.setModel(cbModel);
```

En este caso, se parte de que **lstProvincias** es un objeto de tipo `List<Provincia>` que contiene todos los registros que se necesiten de la tabla y que **cbProvincias** es el `JComboBox` que los va a visualizar. Antes de incluir este código, es imprescindible indicar al componente que va a trabajar con elementos de tipo `Provincia` en lugar de `String`. En NetBeans, simplemente hay que acudir a las propiedades del control y, dentro del apartado **Code**, modificar **Type Parameters** con el valor **<Provincia>**.



2. Asociación de tipo a JComboBox

Importante: El control, una vez arrastrado en NetBeans a su contenedor, muestra un modelo de ejemplo con Strings. **Para realizar el cambio de tipo, es imprescindible el borrado de todos sus elementos.**

Existe otra forma que no requiere la utilización de un `ComboBoxModel`, pero sí mantener la modificación de tipo que se acaba de ver. Se trata de usar el método `addItem` propio del `JComboBox`. La diferencia con el anterior, código aparte, es que selecciona por defecto el primer elemento de la lista, mientras que, en el anterior, no selecciona ninguno. En el siguiente código se puede ver un ejemplo recorriendo la lista con un `foreach`. La particularidad de esta codificación es que incluye una expresión lambda para recorrer la lista, aunque se puede hacer con la instrucción `foreach` tradicional:

```
lstProvincias.forEach(p -> cbProvincias.addItem(p));
```

Antes de pasar a la siguiente opción, lo interesante es conocer cómo acceder al elemento seleccionado en el componente. Por ejemplo, supóngase que se quiere mostrar el id de la provincia en una etiqueta cuando se seleccione un elemento en el desplegable. El evento que se utilizaría en este caso es el `ActionPerformed` al ser la acción por defecto de un desplegable.

El código sería similar al siguiente:

```
private void cbProvinciasActionPerformed(java.awt.event.ActionEvent evt)
{
    JComboBox cb = (JComboBox)evt.getSource();
    Provincia provSeleccionada = (Provincia) cb.getSelectedItem();
    lblIdCombo.setText(String.valueOf(provSeleccionada.getId()));
}
```

En primer lugar, llama la atención que se reference el cuadro combinado de forma indirecta, atendiendo al objeto `ActionEvent`. Aquí no sería estrictamente necesario ya que se conoce su nombre (`cbProvincias`), pero es posible que en algún momento se utilice un manejador genérico para varios `JComboBox`, por lo que podría ser interesante referenciarlo de esta forma.

El elemento seleccionado en un `JComboBox` se obtiene mediante `getSelectedItem()` pero este método devuelve un `Object` y en este caso se necesita una `Provincia`. Por tanto, es necesaria una conversión de tipos para poder conseguir la provincia seleccionada.

Como ya se ha citado, estas dos opciones requieren sobrescribir `toString()` y es posible que esta opción no se quiera tomar porque se podría necesitar otro tipo de implementación para una tarea diferente. En este caso, se puede mantener que la clase utilizada por el `JComboBox` sea un `String` (la que tiene por defecto) y recoger los elementos en una colección que esté coordinada con los elementos del control. Primeramente, fuera del método que cargue los valores en el combo, se declara un mapa:

```
Map<String, Provincia> mapProvincias = new HashMap<String, Provincia>();
```

Posteriormente, se llena dicho mapa recorriendo la lista. Como clave, se usa el nombre de la provincia y el valor corresponde al objeto relacionado.

```
lstProvincias.forEach(p-> {
    mapProvincias.put(p.getNombre(), p);
    cbProvinciasStr.addItem(p.getNombre());
});
```

Ahora, el elemento seleccionado se puede obtener buscando la clave correspondiente en el mapa/diccionario:

```
JComboBox cb = (JComboBox)evt.getSource();
Provincia provSeleccionada = mapProvincias.get(cb.getSelectedItem());
lblIdCombo.setText(String.valueOf(provSeleccionada.getId()));
```

3.2. HERRAMIENTAS PARA EL DESARROLLO DE COMPONENTES VISUALES

A la hora de desarrollar elementos visuales para una interfaz gráfica es necesario disponer de herramientas que permitan la edición de imágenes. En función del tipo de acciones que sea necesario realizar sobre el componente, será recomendable usar herramientas profesionales u otras más sencillas. A continuación, se analizarán algunas de las disponibles en la actualidad.

- [Gimp](#). Programa de distribución gratuita para tareas tales como retoque fotográfico, composición y creación de imágenes. Se puede usar como un programa de edición muy simple para realizar un retoque fotográfico de calidad experta, conversión de formatos de imagen, etc.
- [Adobe Photoshop](#). Uno de los programas para edición de imágenes más populares y reconocidos hoy en día. Una de las funcionalidades más importantes que incorpora esta aplicación es el uso de capas mediante el cual es posible aplicar a la imagen multitud de efectos y tratamientos. Algunas de sus características importantes son:
 - Muy adecuado para el entorno profesional
 - Permite elaborar diseños desde cero.
 - Su sistema de capas permite crear imágenes muy atractivas.
 - Es compatible con gran cantidad de tipos de formatos de imágenes.
 - Dispone de sistemas de filtros, efectos, eliminación de ruido, retoque de imagen, etc.
- [Inkscape](#). Herramienta de diseño gráfico de código abierto que se especializa en la creación y edición de gráficos vectoriales. Se puede usar para diseñar iconos, logotipos y otros elementos gráficos insertables en la interfaz de usuario.
- [Adobe Illustrator](#). Herramienta de Adobe que se utiliza para crear gráficos vectoriales. Es especialmente útil para diseñar iconos y elementos gráficos que deben escalarse sin pérdida de calidad.
- [CorelDRAW](#). Alternativa a Illustrator que permite la creación de gráficos vectoriales de alta calidad. Es popular entre los diseñadores gráficos.
- [Sketch](#): Herramienta de diseño de interfaz de usuario (UI) y diseño de experiencia de usuario (UX) que se utiliza ampliamente en el diseño de aplicaciones y sitios web. Es especialmente útil para diseñar interfaces de usuario modernas y atractivas.
- [Affinity Designer](#). Herramienta de diseño gráfico vectorial que ofrece características similares a Adobe Illustrator. Es una opción económica y de alta calidad.
- [Corel Vector](#). Herramienta de diseño gráfico basada en la web que permite la creación de gráficos vectoriales y es adecuada para diseñadores que trabajan en línea.

3.3. PROPIEDADES, ATRIBUTOS Y MÉTODOS.

Un componente software no deja de ser una clase que se va a tratar de una forma un poco diferente, así que como todas se compone de un conjunto de elementos de tipos básicos a los que se llama atributos, pero si estos cumplen ciertas características pasarán a ser propiedades. Por ejemplo, una etiqueta es un componente ya que 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 se le pueden 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.



3. Propiedad de un componente

Las propiedades de un componente pueden examinarse y modificarse mediante métodos getter/setter, que acceden a dicha propiedad y ya se vieron anteriormente. Si una propiedad no incluye el método set entonces es una propiedad de sólo lectura.

Por ejemplo, si se está generando un componente para crear un botón circular con sombra, se puede tener, entre otras, una propiedad de ese botón que permita cambiar su color, la cual tendría asociados los siguientes métodos:

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

Ejemplo 1: Cuadro de texto con colores de fondo y texto:

Volviendo al ejemplo indicado antes se ve claramente que se trata de un componente que permite la entrada de texto. Así pues, habría que extender el componente de **JTextField**. Por otra parte, el componente tiene dos propiedades, una de tipo Color y otra de tipo entero. En este momento ya se está en disposición de escribir el esqueleto del componente, cumpliendo las condiciones que debe tener para ser un **JavaBean**.

```
import java.awt.Color;
import java.io.Serializable;
import javax.swing.JTextField;
import javax.swing.event.DocumentEvent;
import javax.swing.event.DocumentListener;
public class ComponenteEjemplo extends JTextField implements Serializable {
    /* Color de fondo cuando se llegue al número de caracteres
       indicado en numCaracteres */
    private Color colorFondo;
    private int numCaracteres;
    public ComponenteEjemplo() {
    }
    public Color getColorFondo() {
        return colorFondo;
    }
    public void setColorFondo(Color colorFondo) {
        this.colorFondo = colorFondo;
    }
    public int getNumCaracteres() {
        return numCaracteres;
    }
    public void setNumCaracteres(int numCaracteres) {
        this.numCaracteres = numCaracteres;
    }
}
```

Como se puede ver en el trozo de código anterior, el componente ya cumple las condiciones para ser un JavaBean. Sólo con ese trozo de código, ya se podría añadir el componente a la paleta de componentes de NetBeans, e insertar el componente en cualquier proyecto (aunque todavía no tendría ninguna funcionalidad). Para ello, hay que seguir los siguientes pasos:

1. Compilar el proyecto del componente (botón derecho sobre el proyecto, y elegir *Clean and Build*).
2. Añadir el componente a la paleta. Para hacerlo, se selecciona la clase con el componente y con el botón derecho elegir *Tools > Add to palette....* Aparecerá una ventana para elegir la categoría en la que se quiere incluir el componente y pulsar Aceptar.

Una vez realizados estos pasos, se puede crear un proyecto de prueba para introducir el componente y ver que se pueden editar las propiedades sin hacer nada más. Es importante tener en cuenta que NetBeans es capaz de mostrar y permitir la edición de propiedades automáticamente sólo para los tipos básicos (int, double, String, Color, boolean). En el caso de que la propiedad sea de un tipo más complejo, habrá que realizar un editor de propiedades personalizado.

Ejemplo 2: Panel con imagen de fondo (JPanelImagen)

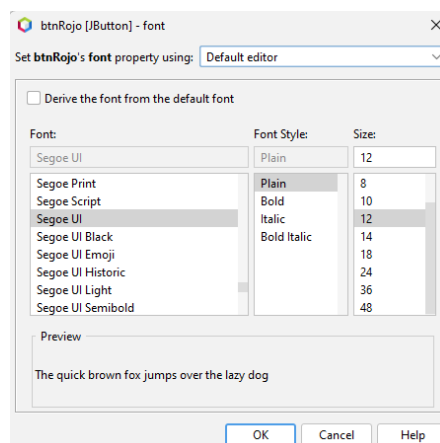
En cuanto al ejemplo del panel con imagen, se le va a añadir una propiedad que recoja la ruta de la imagen que se va a colocar de fondo. No solo hay que incluir en el código la variable miembro privada, sino que para que sea reconocida dentro del JavaBean, hay que crear sus getter/setter con el formato preciso, es decir, **getNombreVariable**, **setNombreVariable** teniendo en cuenta que el nombre de la variable empieza por minúscula. El código quedaría así:

```
private String rutaImagen;  
public String getRutaImagen() {  
    return rutaImagen;  
}  
public void setRutaImagen(String rutaImagen) {  
    this.rutaImagen = rutaImagen;  
}
```

Ahora ya se puede añadir el componente a la paleta. Una opción es seleccionarlo con el botón derecho y elegir Tools > Add to Palette... Se puede elegir una de las categorías existentes (p.e. Beans) o crear una nueva. Si ahora se abre cualquier proyecto de aplicación Java, podrá verse el nuevo control listo para integrarlo en cualquier otro control contenedor y se mostrará la nueva propiedad en su panel correspondiente.

3.3.1. EDITORES DE PROPIEDADES

Una de las principales características de un componente consiste en 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, Font y File, aunque no podrá hacerlo si el tipo de datos de la propiedad es algo más complejo, por ejemplo, si se usa otra clase, como Cliente. Para poder hacerlo habrá que crear nuestro propio editor de propiedades. Ejemplos de este tipo de editores son el diálogo de tipo de letra o de elección de color.



4. Edición de propiedades de tipo de letra.

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**, **Font** y **File**.

Una vez que estén todos los editores, hay que empaquetar las clases con el componente para que use el editor que se ha creado cada vez que necesite editar la propiedad. Así se consigue que cuando se añada un componente en un panel y se seleccione, aparezca una hoja de propiedades, con la lista de las 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.

Los editores de propiedades tienen dos propósitos fundamentalmente:

- Convertir el valor desde y hacia un tipo cadena para ser mostrado adecuadamente conforme a las características de la propiedad
- Validar los datos nuevos cuando son introducidos por el usuario.

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

- Crear una clase que extienda a **PropertyEditorSupport**.
- Añadir los métodos **getAsText** y **setAsText**, que transformarán el tipo de dato de la propiedad en cadena de caracteres o viceversa (estos métodos son opcionales, si no se incluyen no se podrá editar la propiedad escribiendo simplemente texto, habrá que utilizar el panel diseñado para ello).
- Añadir el resto de los métodos necesarios para la clase.
- Asociar el editor de propiedades a la propiedad en cuestión.

Ejemplo 1: Cuadro de texto con colores de fondo y texto:

Se va a probar esta característica en un ejemplo de nuevo componente similar al anterior. Se busca crear un componente que permita introducir un texto. El componente tendrá una propiedad denominada **colorComponente**, que permitirá establecer el color de fondo del componente y el color del texto del componente cuando el usuario haya introducido un

número de caracteres superior al número indicado por una segunda propiedad de tipo entero. Como se puede observar en el enunciado anterior, la propiedad **colorComponente**, es una propiedad compleja, formada por dos colores. En una situación como ésta, no queda otra opción que realizar un editor personalizado, de manera que cuando el programador que use este componente vaya a editar la propiedad **colorComponente**, aparezca un interfaz adecuado para seleccionar dos colores. A continuación, se ve paso por paso como se haría. En primer lugar, hay que crear la clase que va a almacenar la propiedad personalizada. En este problema hay que almacenar dos colores. La clase para la propiedad podría tener esta forma:

```
import java.awt.Color;
import java.io.Serializable;
public class ClaseDosColores implements Serializable {
    private Color colorFondo;
    private Color colorTexto;
    public ClaseDosColores(Color colorFondo, Color colorTexto) {
        this.colorFondo = colorFondo;
        this.colorTexto = colorTexto;
    }
    public Color getColorFondo() {
        return colorFondo;
    }
    public Color getColorTexto() {
        return colorTexto;
    }
}
```

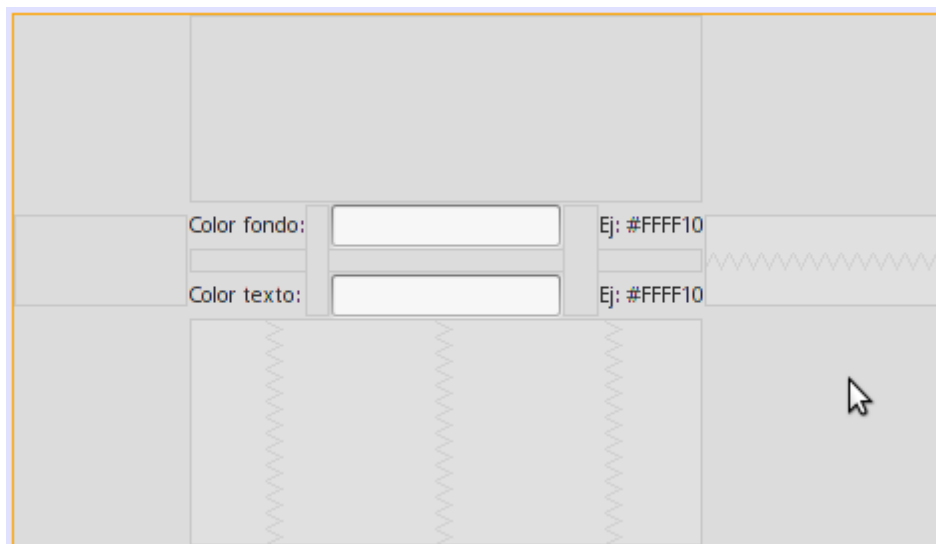
A continuación, se creará el nuevo componente el cual, de forma similar a **ComponenteEjemplo1**, heredará de un **JTextField** y contendrá una propiedad colores para recoger la información de la nueva clase:

```
import java.io.Serializable;
import javax.swing.JTextField;
public class ComponenteEjemplo2 extends JTextField implements Serializable {
    private ClaseDosColores colores;
    private int numCaracteres;
    public ComponenteEjemplo2() {}
    public ClaseDosColores getColores() {
        return colores;
    }
    public void setColores(ClaseDosColores colores) {
        this.colores = colores;
    }
    public int getNumCaracteres() {
        return numCaracteres;
    }
    public void setNumCaracteres(int numCaracteres) {
        this.numCaracteres = numCaracteres;
    }
}
```


Se puede ver que al igual que en **ComponenteEjemplo1**, el componente sigue cumpliendo las condiciones para ser un **JavaBean**.

Es importante observar que la clase en la que se define la propiedad también implementa **Serializable**. Es necesario que lo haga, ya que si no el componente, al incluir una propiedad de este tipo, dejaría de ser **Serializable** y por tanto no cumpliría las condiciones para ser un **JavaBean**.

Si se agrega el componente según está a la paleta, se podría editar la propiedad **numCaracteres**, pero sería imposible editar la propiedad colores. En este siguiente paso, se va a diseñar el panel que va a servir para introducir el valor de la propiedad en tiempo de diseño con una forma similar a la de la figura:



5. Panel para recoger los colores del componente

Para no complicar la introducción de los colores, se recogerán estos por su código en hexadecimal. Es importante indicar que el panel no debe tener botones de aceptar/cancelar ni ningún otro elemento extra que no sea lo estrictamente necesario para recoger el valor de la propiedad. El panel se puede llamar **ColoresPanel**.

Dentro del panel, hay que crear un método que se usará posteriormente para devolver el valor de la propiedad introducido por el desarrollador.

```
public ClaseDosColores getPropiedadSeleccionada() {  
    if (!txtColorFondo.getText().equals("") && !txtColorTexto.getText().equals("")) {  
        Color colorFondo = Color.decode(txtColorFondo.getText());  
        Color colorTexto = Color.decode(txtColorTexto.getText());  
        return new ClaseDosColores(colorFondo, colorTexto);  
    } else {  
        return null;  
    }  
}
```


Es preciso añadir un import a **java.awt.Color** dentro de esta clase. También es importante destacar que este método tiene que devolver una clase del mismo tipo que se haya usado para definir la propiedad que se está editando.

Después de diseñar un panel capaz de recoger la propiedad, hay que crear una clase que extienda de **PropertyEditorSupport**. Esta es la clase que va a indicar a NetBeans que hay un editor de propiedades personalizado y cómo usarlo. Se verá una implementación de esta clase para este ejemplo concreto y a explicar cada uno de sus métodos:

```
import java.awt.Color;
import java.awt.Component;
import java.beans.PropertyEditorSupport;
public class ColoresPropertyEditorSupport extends PropertyEditorSupport {
    private ColoresPanel coloresPanel = new ColoresPanel();
    @Override
    public boolean supportsCustomEditor() {
        return true;
    }
    @Override
    public Component getCustomEditor() {
        return coloresPanel;
    }
    @Override
    public Object getValue() {
        return coloresPanel.getPropiedadSeleccionada();
    }
    @Override
    public String getJavaInitializationString() {
        Color colorFondo = coloresPanel.getPropiedadSeleccionada().getColorFondo();
        Color colorTexto = coloresPanel.getPropiedadSeleccionada().getColorTexto();
        return "new componenteejemplo.ClaseDosColores(new java.awt.Color("
            + colorFondo.getRGB() + "),new java.awt.Color(" + colorTexto.getRGB()
+ "))";
    }
}
```

Explicación de los métodos:

- **supportsCustomEditor**. Este método será llamado por NetBeans para preguntar si existe o no un editor de propiedades personalizado. Se debe devolver true en el caso de que lo tengamos.
- **getCustomEditor**. Este método devolverá el panel que se ha creado para editar la propiedad. Es decir, NetBeans preguntará al método supportsCustomEditor si hay un editor personalizado, y en caso de que lo haya, llamará a este método para obtener el panel que tendrá que mostrarle al usuario. Como se puede ver en la implementación, se devuelve una instancia del panel.
- **getValue**. Una vez que NetBeans muestra el panel para permitir al desarrollador la edición de la propiedad y se pulsa el botón Aceptar, se llamará a este método para

obtener el valor de la propiedad del panel. Es muy importante que este método devuelva un objeto del tipo de la propiedad (**ClaseDosColores** en este caso).

- **getJavaInitializationString**. Este es el método más complicado de todos. Sirve para ayudar al NetBeans en la generación de código necesaria para inicializar la propiedad en tiempo de ejecución. Cuando se arrastra un componente a un formulario, el IDE genera un trozo de código dentro del método **initComponents**. Si únicamente se arrastra el componente y se incluye en un formulario, generaría una llamada al constructor del componente:

```
componenteEjemplo22 = new componenteEjemplo.ComponenteEjemplo2();
```

Si además de arrastrar el componente, se cambia el valor de una propiedad en la vista de diseño, generaría una llamada al setter correspondiente para que cuando el programa sea ejecutado, esa propiedad tenga el valor deseado por el programador. Supóngase que se cambia la propiedad **text**:

```
componenteEjemplo22.setText("Hola");
```

En el caso de que la propiedad editada sea la que se acaba de ver, tendría que funcionar de la misma manera y generar una llamada al setter para inicializar el valor de la propiedad según lo que se haya seleccionado en el panel. El problema está en que el setter de la propiedad recibe una clase creada para ello (**ClaseDosColores**) y por supuesto el NetBeans no tiene ni idea de cómo inicializarla a partir de la información del panel; por tanto, habrá que ayudarlo. El método **getJavaInitializationString** devuelve un String (la parte subrayada en amarillo abajo) con el código que habría que meter entre los paréntesis de la llamada al setter. Concretamente en este caso, NetBeans generaría algo de este estilo:

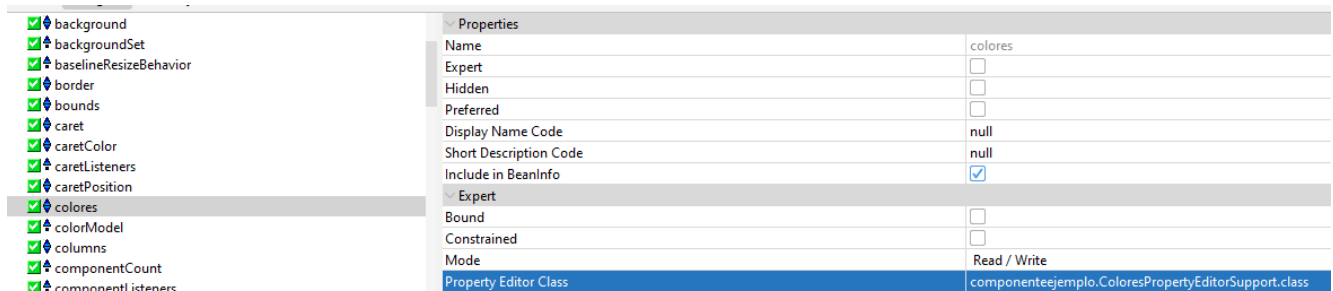
```
componenteEjemplo22.setColores(new componenteEjemplo.ClaseDosColores(new  
java.awt.Color(-65281),new java.awt.Color(-16711936)));
```

Como se puede observar en el código anterior, se ha utilizado el código RGB del color para inicializar los dos colores seleccionados en el panel. Desgraciadamente, no existe un mecanismo sencillo para orientar sobre cómo programar este método. Simplemente hay que conseguir crear un String que encaje perfectamente en el setter de la propiedad y que sea capaz de inicializar el valor de la propiedad según lo que se haya seleccionado en el editor de propiedades.

De esta forma, una vez arrancado el proyecto de prueba, el valor de la propiedad tomaría los valores deseados y dentro del componente se podría trabajar con esos valores para programar la funcionalidad deseada.

El último paso en la creación del editor de propiedades personalizado consiste en crear el **BeanInfo** del componente, lo que sirve para indicarle a NetBeans a que propiedad corresponde el **PropertyEditorSupport** que se ha creado.

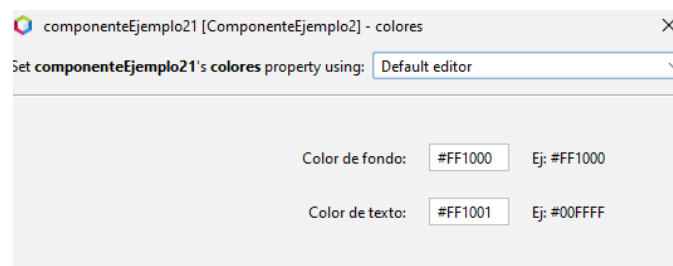
Para ello, hay que situarse encima del componente y con el botón derecho seleccionar **BeanInfo Editor**. De esta forma aparecerá una nueva clase en el proyecto. Al abrirla y situarse en vista de diseño se busca la propiedad recién creada (colores en este caso), e indicar la clase `PropertyEditorSupport` que se ha hecho anteriormente con nombre completo de paquete y extensión `.class`



6. Vista de diseño del BeanInfo del componente.

Es importante que se escriba perfectamente el nombre de la clase, con su paquete incluido, y terminando en `.class`. Si no se hace así, no mostrará ningún error, pero el editor personalizado no aparecerá. Otro detalle importante es que el BeanInfo debe ser regenerado si se cambia la definición de nuestro componente (si se añaden o quitan propiedades o métodos).

Después de los pasos anteriores, solo queda recompilar el proyecto (Clean and build) y agregar el componente a la paleta. Si se agrega el componente a un proyecto de prueba, se verá como la propiedad puede ser editada con nuestro editor personalizado.



7. Propiedad ajustada desde editor

Con los pasos anteriores es suficiente para agregar un editor de propiedades a un componente. Solo quedaría incluir la funcionalidad al componente.

Ejemplo 2: Panel con imagen de fondo (JPanelImagen)

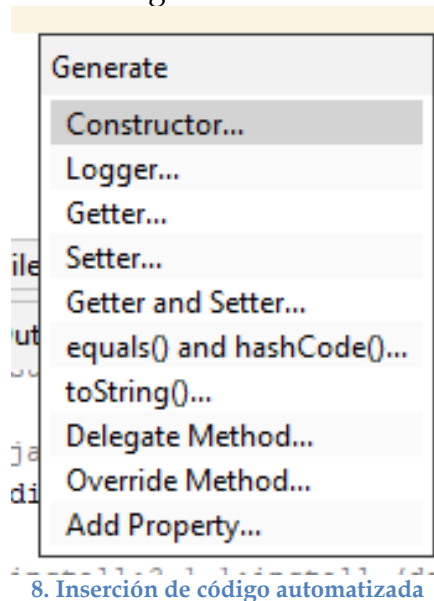
Para seguir trabajando con propiedades, se verá el control del ejemplo del panel con imagen de fondo. En primer lugar, hay que modificar el tipo de la variable `rutaImagen` a la vez que se deben regenerar los getters y setters.

Como lo que se hará es almacenar la información de un fichero, se utilizará la clase `File`:

```
private File rutaImagen;  
public File getRutaImagen() {  
    return rutaImagen;  
}  
public void setRutaImagen(File rutaImagen) {  
    this.rutaImagen = rutaImagen;  
}
```

Ahora, para probar el control en otro proyecto hay que recompilar seguido de la eliminación del control y su nueva inserción en el proyecto de prueba. Al seleccionar **rutaImagen**, ahora da la opción de seleccionar un fichero de los discos accesibles por el sistema.

Antes de continuar profundizando en los aspectos de este apartado, se va a seguir configurando el control. Al ajustar la ruta de la imagen, hay que pintar esa imagen de fondo en el control. Para ello, es preciso sobrescribir el método `paintComponent` propio de **JComponent**. Una forma rápida de hacerlo es situarse en la zona de código dentro de la clase y con el menú contextual, eligiendo **Insert Code** o con el atajo de teclado `Alt + Insert`, el IDE dará varias opciones de inserción de código:



8. Inserción de código automatizada

En este caso, lo que se debe hacer es sobrecargar dicho método, así que hay que elegir **Override Method**. Automáticamente, el IDE proporciona el siguiente fragmento de código:

```
@Override  
protected void paintComponent(Graphics g) {  
    super.paintComponent(g);  
}
```

Después de algunas modificaciones que se explicarán después, el código del método quedará así:

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    if (rutaImagen != null && rutaImagen.exists()){
        ImageIcon imageIcon = new ImageIcon(rutaImagen.getAbsolutePath());
        g.drawImage(imageIcon.getImage(), 0, 0, null);
    }
}
```

En primer lugar, hay que validar si el objeto `rutaImagen` es distinto de nulo y si el fichero existe. En caso afirmativo, el sistema creará un objeto de tipo [ImageIcon](#) con el gráfico obtenido de la ruta absoluta dada por la propiedad y se dibujará una imagen con ese gráfico, posiciones x e y de valor cero y no se introducirá un objeto observador que representará al objeto al que notificar que la imagen ha sido convertida. Una vez recompilada y reinsertada en un control, se verá que, al darle una imagen desde su propiedad, el control la colocará de fondo.

Pero inicialmente, lo que se deseaba es poder cambiar la opacidad de dicha imagen y para eso se necesita una clase que recoja tanto el fichero como el grado de opacidad, el cual se mide con un tipo **Float** con valores entre 0 y 1.

La clase se denominará **ImagenFondo** y tendrá el siguiente código:

```
import java.io.File;
public class ImagenFondo {
    public ImagenFondo(File rutaImagen, float opacidad) {
        this.rutaImagen = rutaImagen;
        this.opacidad = opacidad;
    }
    private File rutaImagen;
    private float opacidad;
    public File getRutaImagen() {
        return rutaImagen;
    }
    public void setRutaImagen(File rutaImagen) {
        this.rutaImagen = rutaImagen;
    }
    public float getOpacidad() {
        return opacidad;
    }
    public void setOpacidad(float opacidad) {
        this.opacidad = opacidad;
    }
}
```

Para construirla de forma rápida, basta con introducir las dos variables, su import, refactorizar para encapsular en getter/setters e insertar un constructor que reciba ambas variables. Para reflejar estos cambios hay que irse al panel y cambiar el tipo de **rutaImagen** y de paso su nombre.

Todo ello requiere que se eliminen los antiguos getter/setter y que se modifique también la sobrescritura de **paintComponent**, aunque de momento solo manteniendo la funcionalidad referida a la imagen sin tocar la opacidad:

```
public class JPanelImagen extends JPanel implements Serializable {
    private ImagenFondo imagenFondo;
    public JPanelImagen(){}
    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        File rutaImagen = imagenFondo.getRutaImagen();
        if (rutaImagen != null && rutaImagen.exists()){
            ImageIcon imageIcon = new ImageIcon(rutaImagen.getAbsolutePath());
            g.drawImage(imageIcon.getImage(), 0, 0, null);}
    }
    public ImagenFondo getImagenFondo() {
        return imagenFondo;
    }
    public void setImagenFondo(ImagenFondo imagenFondo) {
        this.imagenFondo = imagenFondo;
    }
}
```

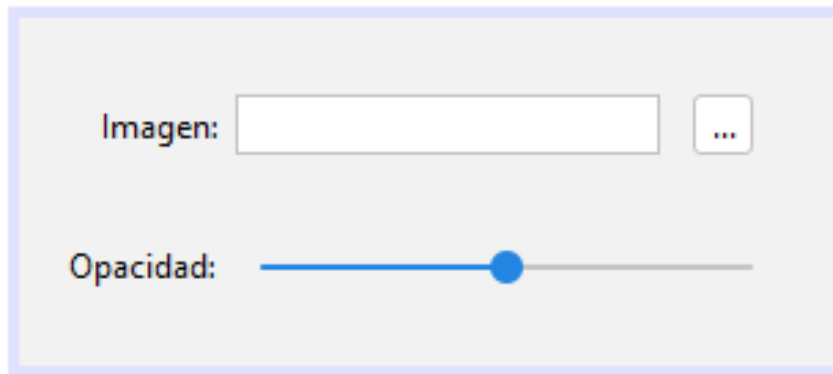
Como puede verse en **paintComponent**, ahora se hace uso de **imagenFondo** y, dado que el método antes trabajaba con el fichero subyacente, ahora lo que se hace es crear una variable interna al método con el mismo nombre (**rutaImagen**) recogiendo la de **imagenFondo**. De esta forma, los cambios de la codificación son mínimos bastando con incluir una sola línea de código.

Anteriormente se vio que tanto componente como clases utilizadas deben ser serializables para poder utilizarlas en un Bean. Por tanto, hay que implementar **Serializable** en la clase **ImagenFondo**:

```
public class ImagenFondo implements Serializable{...}
```

Ahora se necesita diseñar el panel que servirá de soporte como editor de la propiedad recién creada. Por ejemplo, se le puede dar el nombre de la clase subyacente seguido de panel (**ImagenFondoPanel**). Este elemento se añade desde *New > JPanel Form* desde el menú contextual del proyecto.

Su diseño será como sigue pasando a explicar las propiedades a continuación:



9. Editor de propiedades de imagen y opacidad

El editor contiene etiquetas para Imagen y Opacidad, un cuadro de texto (**txtImagen**) para la imagen el cual tendrá su propiedad editable a false, un botón para su selección mediante el fichero (**btnImagen**) y un Slider para la opacidad (**sldOpacidad**).

El siguiente paso consiste en programar el evento de clic al botón para que lance un cuadro de diálogo que permita seleccionar el fichero de imagen. Dentro del manejador del evento del botón, hay que introducir el siguiente código:

```
JFileChooser chooser = new JFileChooser();
int resultado = chooser.showOpenDialog(this);
if (resultado == JFileChooser.APPROVE_OPTION){
    File fichero = chooser.getSelectedFile();
    txtImagen.setText(fichero.getAbsolutePath());
}
```

Lo que se hace aquí es instanciar un [JFileChooser](#) el cual representa a un diálogo que permite seleccionar un fichero del sistema de archivos. Se llama a su método **showOpenDialog** que devuelve un entero siendo este valor la retroalimentación que se recibe del control con la acción del usuario. Por ejemplo, si el usuario ha seleccionado el fichero y lo acepta, dicho valor se corresponde con el de la constante **JFileChooser.APPROVE_OPTION**

Otra acción importante para configurar el editor de propiedades es incluir una propiedad que trabaje con un objeto de la clase subyacente al editor, en este caso, **ImagenFondo**:

```
public ImagenFondo getSelectedValue() {
    File f = new File(txtImagen.getText());
    Float opacidad = sldOpacidad.getValue() / 100f;
    return new ImagenFondo(f, opacidad);
}
```

Este código prácticamente se interpreta solo. Primeramente, se crea un fichero a partir de la ruta recogida en el cuadro de texto; después se recoge la opacidad mediante el Slider, pero como este tiene un valor entero entre 0 y 100 y la opacidad es un decimal entre 0 y 1, hay que dividir por 100.

Como puede verse, se hace una conversión del divisor a float para que la operación sea coherente con el tipo devuelto, ya que, si no se hace sería una división entre dos enteros la cual devuelve también un entero. Por último, se devuelve la imagen instanciada con el constructor que recoge ambos parámetros.

Ahora que se tienen todos los datos es cuando toca modificar el método `paintComponent` para incluir la opacidad. Se muestra a continuación el código que irá dentro del método y se explica después:

```
super.paintComponent(g);
if (imagenFondo != null && imagenFondo.getRutaImagen() != null &&
imagenFondo.getRutaImagen().exists()){
    ImageIcon imageIcon = new
ImageIcon(imagenFondo.getRutaImagen().getAbsolutePath());
    Graphics2D g2d = (Graphics2D) g;
    g2d.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER,
imagenFondo.getOpacidad()));
    g.drawImage(imageIcon.getImage(), 0, 0, null);
    g2d.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_OVER, 1));
}
```

Bien, en este caso se prescinde definitivamente de `rutaImagen` ya que lo primero que hay que comprobar es si `imagenFondo` no es nula para evitar excepciones de tipo `NullPointerException`. Por tanto, en el `if` lo primero que se pregunta es si no lo es. Lo primero que se puede pensar al ver este código es ¿y qué pasa si es nulo y luego pregunto por `getRutaImagen()`? ¿no debería lanzar la excepción? En ese caso ¿no sería un código inseguro? Nada más lejos. La naturaleza de la evaluación de expresiones en Java y otros lenguajes basados en C tiene como base el cortocircuito de expresiones. Esto quiere decir que, en una evaluación, en cuanto se cumpla la condición que la hace verdadera o falsa sin importar lo que venga después, lo siguiente no se evaluará. De este modo, como se están evaluando operadores AND, si `imagenFondo` es nulo, ese primer operando devolvería `false`, con lo que la expresión completa devolvería lo mismo.

El compilador entonces ya no evalúa más porque sabe seguro que devolverá `false`. Esto no solo es útil para simplificar una expresión `if` sin necesidad de anidar otras, sino que es bueno para el rendimiento teniendo en cuenta que no tendrá que computar el resto de la expresión.

Una vez que se cumplen los requisitos para pintar la imagen y su opacidad, se puede ver cómo se ajusta esta mediante código utilizando una conversión de `Graphics` a [Graphics2D](#). Dado que son operaciones de una API muy especializada, no se explicará en detalle, pero puede deducirse fácilmente cómo se ajusta la opacidad en la línea de código correspondiente. Por último, una vez pintada la imagen, se vuelve a ajustar la opacidad por defecto a 1.

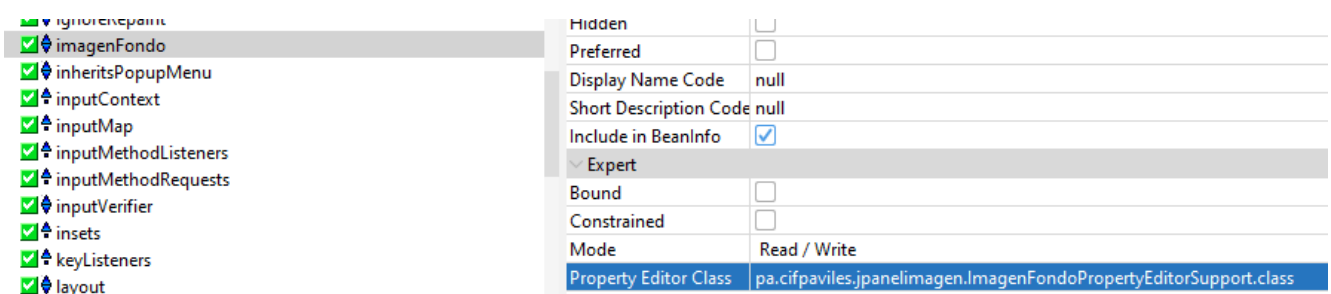
Siguiendo con el proceso, el paso que hay que realizar es crear la clase `PropertyEditorSupport` que sirva de soporte al editor de propiedades. Heredará de **`PropertyEditor`** y habrá que sobrescribir algunos métodos.

En este ejemplo se hará con cuatro, pero puede que se necesiten más en otros casos:

```
public class ImagenFondoPropertyEditorSupport extends PropertyEditorSupport{
    private ImagenFondoPanel pnlImgFondo = new ImagenFondoPanel();
    @Override
    public boolean supportsCustomEditor() {
        return true;
    }
    @Override
    public Component getCustomEditor() {
        return pnlImgFondo;
    }
    @Override
    public String getJavaInitializationString() {
        ImagenFondo imFondo = pnlImgFondo.getSelectedValue();
        return "new pa.cifpaviles.jpanelimagen.ImagenFondo("+
            "new java.io.File(\""+imFondo.getRutaImagen().getAbsolutePath().replace("\\",
            "\\\"\\\"\\\"")+"\")"+", "+
                imFondo.getOpacidad()+")";
    }
    @Override
    public Object getValue() {
        return pnlImgFondo.getSelectedValue();
    }
}
```

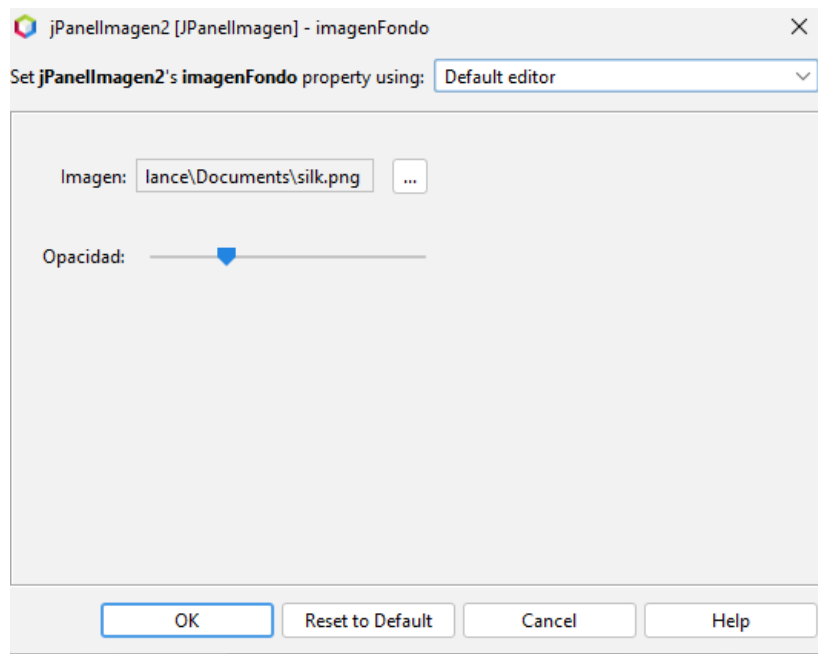
En resumen, se devuelve true para indicar que sí se soporta un editor personalizado, el cual será una instancia del panel creado para la ocasión y el valor seleccionado será el que se diseñó en su momento para el panel. También incluye la cadena de inicialización para que el sistema sea capaz de instanciar al completo el objeto ImagenFondo. Como puede verse, hay que incluir los nombres completos con sus correspondientes paquetes y en este caso concreto, al incluir un nombre de fichero en Windows, escapar no solo las comillas sino la barra invertida ya que se utiliza para construir la ruta del fichero. Con las rutas en sistemas basados en UNIX no se da este problema al hacer uso de la barra de división.

Ahora, con la clase `JPanelImagen`, es decir, el componente diseñado para la ocasión se crea un `BeanInfo` asociado. En la vista de diseño del `BeanInfo` hay que acudir a la propiedad `imagenFondo` y añadir el `PropertyEditorSupport` como cadena con el nombre completo (incluyendo el paquete y la extensión `.class`). En este caso conviene no equivocarse porque, de hacerlo, el sistema no mostrará ningún error, aunque no funcionará el editor:



10. JavaBean con PropertyEditorSupport

Si ahora se introduce en otro proyecto, podrá verse cómo se ajusta la imagen con su opacidad.



11. Editor de propiedades para imagen de fondo.

3.3.2. 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 la cual sea 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 array y se identifica mediante los siguientes patrones de operaciones para leer o escribir elementos individuales del array o el array entero:

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

3.3.3. PROPIEDADES COMPARTIDAS Y REGISTRADAS

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 [EventObject](#)) 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 se vieron 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.

3.4. 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 habrá que hacer lo siguiente:

- Crear una clase para los eventos que se lancen.
- Definir una interfaz que represente el escuchador (listener) asociado al evento. Debe incluir una operación para el procesamiento del evento.
- Definir dos operaciones, para añadir y eliminar listeners. Si se quiere tener más de un listener para el evento habrá que almacenarlos internamente 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.

Algunos tipos de eventos asociados a objetos son:

- [EventObject](#). Clase principal de la que derivan todos los eventos.
- [MouseEvent](#). Eventos relativos a la acción del ratón sobre el componente.
- [ComponentEvent](#). Eventos relacionados con el cambio de un componente de tamaño, posición, etc.
- [ContainerEvent](#). Evento producido al añadir o eliminar componentes sobre un objeto de tipo [Container](#)
- [WindowEvent](#). Este tipo de eventos se produce cuando una ventana ha sufrido algún tipo de variación, desde su apertura o cierre hasta el cambio de tamaño.
- [ActionEvent](#). Evento producido al detectarse la acción sobre un componente. Es uno de los más comunes ya que modela acciones como la pulsación sobre un botón o el marcado de una casilla de selección.

Ejemplo 1: Cuadro de texto con colores de fondo y texto:

Volviendo al componente del primer ejemplo creado anteriormente y pudiéndose ya editar sus propiedades desde la paleta de NetBeans, solo queda programar la funcionalidad. En este

caso concreto, hay que ser capaces de saber en cada momento cuantos caracteres hay introducidos y cambiar el color de fondo en consecuencia. Para ello se escribe el siguiente código en el constructor:

```
private Color colorPorDefecto;  
public ComponenteEjemplo() {  
    super();  
    /* Se salva el color que tendrá el componente por defecto */  
    colorPorDefecto = getBackground();  
    super.getDocument().addDocumentListener(new DocumentListener() {  
        @Override  
        public void insertUpdate(DocumentEvent e) {  
            analizaContenido();  
        }  
        @Override  
        public void removeUpdate(DocumentEvent e) {  
            analizaContenido();  
        }  
        @Override  
        public void changedUpdate(DocumentEvent e) {  
            analizaContenido();  
        }  
        private void analizaContenido() {  
            if (getText().length() >= numCaracteres) {  
                setBackground(colorFondo);  
            } else {  
                setBackground(colorPorDefecto);  
            }  
        }  
    });  
}
```

Como se puede observar en el código anterior, se usan las dos propiedades para modificar el color de fondo del componente según el número de caracteres introducidos en el mismo. Se ha creado también una variable interna para almacenar el color de fondo por defecto y poder restablecerlo en el caso de que el usuario borre caracteres.

3.5. INTROSPECCIÓN. REFLEXIÓN

Cuando se trabaja con cualquier clase, y los componentes no dejan de serlo, el editor proporciona algunas ayudas a la edición que dan pistas acerca de los métodos que se pueden usar o qué argumentos se deben 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.

Como ya se vio anteriormente, también se puede hacer uso de una clase asociada de información del componente (BeanInfo) que describe explícitamente sus características para que puedan ser reconocidas.

3.5.1. REFLEXIÓN

La reflexión es una de las técnicas más flexibles y potentes que posee el lenguaje Java. Para convertirse en un desarrollador Java avanzado, obtener una comprensión sólida de la reflexión (también conocida como [API Core Reflection](#)) es un paso esencial. Se puede definir como un mecanismo para analizar datos sobre un programa e incluso cambiar la estructura y su comportamiento mientras se ejecuta.

En la primera versión de Java, la API Core Reflection no existía, pero ya se podían encontrar métodos en java.lang.Class para acceder a algunos metadatos sobre los tipos de objetos en tiempo de ejecución, como el nombre de la clase. Por ejemplo, código como el siguiente ya funcionaría en Java 1.0:

```
Object o = getObject();  
System.out.println(o.getClass().getName());
```

La siguiente versión de Java, la versión 1.1, introdujo por primera vez capacidades de reflexión completas. El paquete java.lang.reflect pasó a formar parte de la distribución estándar y permitió a los desarrolladores acceder a una gama completa de operaciones reflectantes. Esto incluía cosas como llamar a métodos sobre objetos de tipo desconocido que habían sido creados de forma reflexiva.

Hay que en cuenta que la API Core Reflection es anterior a la llegada de la API de colecciones de Java (que apareció en Java 1.2). Como resultado, tipos como List no aparecen en la API Core Reflection; en su lugar, se utilizan matrices. Esto hace que algunas partes de la API sean más complicadas de usar de lo que serían de otro modo.

A continuación, se enumeran algunas tareas que se pueden hacer mediante reflexión:

- Crear un objeto reflexivamente.
- Localizar un método en el objeto.
- Llamar al método localizado.

Esas tareas se logran con código como el siguiente. En este primer ejemplo, el manejo de excepciones se ignora, por ahora, para mejorar la claridad del código:

```
Class<?> selfClazz = getClass();  
Constructor<?> ctor = selfClazz.getConstructor();  
Object o = ctor.newInstance();  
Method toString = selfClazz.getMethod("toString");  
Object str = toString.invoke(o);  
System.out.println(str);
```

El código comienza a partir de un objeto de clase, que se obtuvo mediante `getClass()` en este ejemplo. En la práctica, este objeto de clase también se puede obtener de un cargador de clases o, más raramente, como un literal de clase (como `String.class`). La reflexión se puede combinar con la carga de clases para proporcionar a los programas Java la capacidad de trabajar con código que era completamente desconocido en el momento de la compilación. Este enfoque es la base de las arquitecturas de complementos y otros mecanismos muy dinámicos que Java puede utilizar.

No importa cómo se haya obtenido, una vez que se tenga un objeto de clase, se puede llamar a `getConstructor()` en él. Este método puede necesitar un número variable de argumentos para hacer frente a la posibilidad de múltiples constructores para la clase.

El ejemplo anterior llama al constructor predeterminado (nulo); este constructor no acepta argumentos. El resultado es un objeto constructor del cual se puede obtener una instancia de la clase, como si se hubiera llamado directamente al constructor.

Desde el objeto constructor, llamar a `newInstance()` hace que se cree un nuevo objeto del tipo apropiado. Este método por defecto necesitaría proporcionar los parámetros correctos a la llamada.

Por cierto, el objeto de clase también declara un método `newInstance()` que se puede utilizar para crear nuevos objetos directamente, sin crear un objeto constructor intermedio. (`Class.newInstance()` está en desuso debido a su comportamiento de manejo de excepciones, lo que se verá posteriormente).

También es posible llamar a `getMethod()` para recuperar un objeto de método del objeto de clase. Esto representa la capacidad de llamar a un método que pertenece a la clase desde la que se creó el objeto del método. Una vez que se tenga el objeto Método, puede usar `invoke()` para llamar al que representa.

Los métodos de instancia deben llamarse de manera reflexiva con el objeto receptor (el objeto al que se invoca el método) como argumento inicial para `invoke()`, seguido de cualquier parámetro del método. En el ejemplo anterior, "o" es el receptor y no hay parámetros para la llamada a `toString()`.

También es posible acceder a los campos de forma reflexiva; la API y la sintaxis son muy similares al caso de los métodos. La clase principal es `java.lang.reflect.Field` y se puede acceder a los campos a través de `Class.getField()` y otros métodos similares.

A continuación, se puede ver un código ligeramente más complejo donde constructores y métodos requieren argumentos:

```
public record Person(String firstName, String lastName, int age) {}

Class<?> selfClazz = Class.forName("pa.cifpaviles.dam.reflectionexample.Person");
Constructor<?> ctor = selfClazz.getConstructor(String.class, String.class,
int.class);
Object o = ctor.newInstance("Robert", "Smith", 63);
Method toStr = selfClazz.getMethod("equals", Object.class);
Object other = new Object();
Object isEqual = toStr.invoke(o, other);
System.out.println(isEqual);
```

Antes de explicar los aspectos de reflexión en el código, conviene conocer en qué consiste la primera línea. Se trata de un elemento introducido a partir del JDK 14 conocido como Java Record. Estos nuevos elementos representan lo que se podría considerar un registro. Son muy habituales cuando se necesita trabajar Data Objects (objetos enlazados a orígenes de datos). Como se puede observar, tienen una especie de método constructor que recoge las tres propiedades del registro. Por tanto, para el ejemplo de código, si se quisieran crear elementos registro de `Person`, se escribiría un código similar al siguiente:

```
Person p = new Person("Robert", "Smith", 63);
```

El acceso a sus propiedades es igual que en una clase teniendo en cuenta que no hay getter/setters:

```
System.out.println(p.age());
System.out.println(p.firstname());
System.out.println(p.lastname());
```

Volviendo al código, las llamadas a `getConstructor()` y `getMethod()` cuentan con varios objetos de clase. Estos representan la firma de tipo de los constructores o métodos.

Las llamadas a `Constructor.newInstance()` e `invoke()` están, de forma similar, provistas de objetos que son los parámetros de las llamadas. Estos parámetros, por supuesto, deberían coincidir con los tipos esperados de argumentos. El sistema de tipos y la naturaleza de Java entran en juego en varios lugares dentro de las API reflectantes. Por tanto, es preciso preocuparse de:

- Tratar con excepciones
- Sobrecarga
- Control de acceso

Cada uno de estos elementos puede complicar la escritura de código Java reflexivo.

El tratamiento de excepciones es el primer obstáculo a solventar, aunque si se trabaja con un IDE, este puede ayudar a gestionar las citadas excepciones.

La existencia de sobrecarga de métodos en el lenguaje Java implica que cuando se busca un método (o constructor) mediante `getMethod()`, pasar el nombre no es suficiente; en su lugar, también se debe proporcionar la firma del método, representada como una secuencia de objetos de clase (que se convertirán en una `Class[]`). Se puede considerar el uso de `Class[]` (y `Object[]` cuando un método se llama de forma reflexiva) como una especie de defecto de diseño, que hace que sea más difícil trabajar con la API. En realidad, esto es solo un desafortunado accidente porque la API Core Reflection se agregó antes de que existieran las bibliotecas de Colecciones, por lo que debe mantenerse en su forma original por razones de compatibilidad con versiones anteriores.

Otro problema se relaciona con el control de acceso: la API proporciona dos métodos diferentes, `getMethod()` y `getDeclaredMethod()`, para acceder a los métodos de forma reflexiva. La primera de estas dos posibilidades, `getMethod()`, se utiliza para buscar métodos públicos. Por el contrario, `getDeclaredMethod()` se puede utilizar para encontrar cualquier método declarado en una clase, incluso métodos privados.

De forma predeterminada, el código reflexivo aún respeta la semántica de control de acceso del lenguaje Java, pero es posible anular esa semántica porque la API proporciona el método `setAccessible()`, que se puede invocar en métodos, constructores y campos. Una vez que se haya llamado a este método en un objeto accesible, se ignorarán los modificadores de control de acceso.

El método `setAccessible()` se puede considerar peligroso, ya que permite a los programadores desactivar selectivamente partes del sistema de control de acceso cuando están trabajando de forma reflexiva. Representa un compromiso en el subsistema reflexivo: a veces simplemente no hay otra forma de obtener el acceso necesario, pero cuando se usa en exceso puede causar todo tipo de problemas de seguridad.

Se puede argumentar que el compromiso no es tan malo, porque cuando se obtiene un objeto método u otro objeto accesible, la clase correspondiente definitivamente ya se ha cargado, lo que significa que el código de bytes de la clase ya ha pasado la verificación por parte del momento en que se haga la invocación reflexiva. Sin embargo, usar `setAccessible()` para obtener acceso a métodos que de otro modo serían inaccesibles representa una violación de la encapsulación y un uso que el autor del código original no pretendía.

3.6. PERSISTENCIA DEL COMPONENTE

A veces, se necesita 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 proceso de recuperación se le conoce como deserialización.

Para que un componente persista, siempre desde el punto de vista Java, se deben implementar los interfaces [java.io.Serializable](#) o [java.io.Externalizable](#) los cuales 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 esta interfaz hay que 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. Se utilizará el modificador transient para especificar los campos que no se quieren 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.

El estado de un objeto viene dado, básicamente, por el estado de sus campos. Así, serializar un objeto consiste, básicamente, en guardar el estado de sus campos. Si el objeto a serializar tiene campos que a su vez son objetos, habrá que serializarlos primero. Éste es un proceso recursivo que implica la serialización de todo un grafo (en realidad, un árbol) de objetos. Además, también se almacena información relativa a dicho árbol, para poder llevar a cabo la reconstrucción del objeto serializado.

Un ejemplo de uso consiste en utilizar los mecanismos de serialización disponibles para serializar un objeto guardándolo en un fichero y para realizar el proceso inverso, recuperándolo desde el fichero.

```
FileOutputStream fos = new FileOutputStream("fichero.bin");
FileInputStream fis = new FileInputStream("fichero.bin");
ObjectOutputStream out = new ObjectOutputStream(fos);
ObjectInputStream in = new ObjectInputStream(fis);
ClaseSerializable o1 = new ClaseSerializable();
ClaseSerializable o2 = new ClaseSerializable();
/* Escribir el objeto en el fichero */
out.writeObject(o1);
out.writeObject(o2);
. . .
/* Leer el objeto del fichero (en el mismo orden !!) */
o1 = (ClaseSerializable)in.readObject();
o2 = (ClaseSerializable)in.readObject();
```

En ocasiones puede interesar tomar el control sobre el proceso de serialización de una clase en concreto. Esto se puede hacer 'sobrecargando' los métodos **writeObject** y **readObject** de la clase cuya serialización se quiere controlar. En realidad, no se puede hablar de sobrecarga, puesto que estos métodos no están definidos en **java.lang.Object**.

Para 'personalizar' la serialización de un objeto, basta añadir un método tal que:

```
private void writeObject (ObjectOutputStream stream) throws IOException{
    stream.defaultWriteObject();
    . . .
}
```

Es necesario respetar exactamente tanto la signatura del método como la primera acción a realizar. A continuación, pueden añadirse otras acciones que escriban en el **stream** dado. También será necesario añadir un método para hacer el paso inverso:

```
private void readObject (ObjectInputStream stream) throws IOException{
    stream.defaultReadObject();
    . . .
}
```

3.7. PRUEBA DE LOS COMPONENTES

Las pruebas visuales son un aspecto crucial del proceso de desarrollo, que implica la evaluación sistemática y rigurosa de los componentes de la interfaz de usuario (UI), la disposición, el diseño, la interactividad, la capacidad de respuesta y la estética general de una aplicación web o móvil. Su objetivo es identificar y rectificar inconsistencias visuales, desviaciones en las especificaciones de diseño y cualquier problema de usabilidad que pueda surgir debido a los diferentes entornos de visualización, sistemas operativos y dispositivos.

Algunas de las herramientas de prueba y bibliotecas más extendidas para su uso con Java son:

- [JUnit](#). Es el marco de pruebas unitarias de Java más popular. De código abierto, se utiliza para escribir y ejecutar pruebas automatizadas repetibles.
- [Selenium](#). Entorno de pruebas que se utiliza para comprobar si el software que se está desarrollando funciona correctamente. Esta herramienta permite: grabar, editar y depurar casos de pruebas que se pueden automatizar. Lo más atractivo de Selenium es que se pueden editar acciones o crearlas desde cero. Esta herramienta también ayuda mucho en las pruebas de regresión porque consigue pruebas automatizadas que luego se pueden reutilizar cuando se necesite.
- [TestNG](#). Es un marco de automatización de pruebas para Java. Se basa en NUnit y JUnit, añadiendo nuevas funcionalidades para hacerlo más usable y poderoso. En este sentido, las siglas NG significan «Next Generation», próxima generación. El objetivo principal era crear un nuevo marco de trabajo que resolviera los fallos de JUnit. Se trata de una herramienta de código abierto y gratuita.

Entre los tipos de pruebas que se pueden realizar sobre componentes visuales cabe destacar:

- Pruebas de funcionalidad. Sirven para verificar que los componentes se comportan según lo esperado; por ejemplo, probar si un botón realiza la acción correcta.
- Pruebas de usabilidad. Evalúan la facilidad de uso y la experiencia de usuario.
- Pruebas de rendimiento. Miden el rendimiento de los componentes en términos de tiempo y eficiencia.
- Pruebas de accesibilidad. Sirven para asegurarse de que los componentes sean accesibles para personas con diversidad funcional.

3.8. EMPAQUETADO DE COMPONENTES

Una vez creado el componente, es necesario empaquetarlo para poder distribuirlo y utilizarlo después. Para ello se necesitará 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.



12. Empaquetado de componentes

Se pueden incluir varios componentes en un mismo archivo. El paquete jar debe incluir un fichero de manifiesto (con extensión .mf) que describa su contenido.

La forma más sencilla de generar el archivo jar es utilizar la herramienta Clean and Build del proyecto en NetBeans que deja el fichero .jar en el directorio target del proyecto, aunque siempre se puede recurrir a la orden jar y crearlo directamente:

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

Una vez que se tiene un componente Java empaquetado es muy sencillo añadirlo a la paleta de componentes gráficos de NetBeans, tal como se vio anteriormente.

En caso de proyecto Maven, el empaquetado se complica un poco al no quedar bien configurado el archivo de manifiesto. Existen dos formas de resolver esta situación. Una manual añadiendo la clase principal y todas las dependencias al archivo de manifiesto, pero es que, además, estas deben estar situadas en un directorio lib, con lo que el despliegue se complica al tener que hacer varias operaciones de forma manual. Por suerte, y dada la naturaleza de Maven, se puede realizar un despliegue de forma mucho más automatizada con solo incluir unas líneas en el archivo pom.xml.

Las líneas a incluir son las siguientes:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <version>3.3.0</version>
      <configuration>
        <archive>
          <manifest>
            <addClasspath>true</addClasspath>
            <classpathPrefix>libs/</classpathPrefix>
            <mainClass>(clase principal con nombre largo de
paquete)</mainClass>
          </manifest>
        </archive>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <version>3.6.1</version>
      <executions>
        <execution>
          <id>copy-dependencies</id>
          <phase>prepare-package</phase>
          <goals>
            <goal>copy-dependencies</goal>
          </goals>
          <configuration>
            <outputDirectory>
              ${project.build.directory}/libs
            </outputDirectory>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

En definitiva, se trata de añadir una sección build en la que se incluirá una sección de plugins donde se alojarán dos: uno para que el manifiesto incluya la clase principal y otro para que se copien las dependencias al directorio lib, de la misma forma que se haría en la modalidad manual.

Una vez hecho esto, basta con hacer doble clic sobre el jar resultante o lanzar la siguiente orden en consola: `java -jar nombreakplicacion.jar`

3.9. CASO PRÁCTICO: CONEXIÓN DE COMPONENTE A SERVICIO REST

En este apartado se muestra cómo se puede llamar a un servicio web desde un componente de usuario Java.

Los servicios web consisten en un sistema de software diseñado para permitir interoperabilidad entre máquinas dentro de una red. En definitiva, se basan en APIs que son publicadas, localizadas e invocadas a través de la web; es decir, una vez desarrolladas, son instaladas en un servidor y otras aplicaciones (u otros servicios web) pueden consumir sus servicios.

Como norma general, el transporte de los datos se realiza a través del protocolo HTTP y la representación de los datos mediante sistemas como XML/JSON. Sin embargo, no hay reglas fijas en los servicios web y en la práctica no tiene por qué ser así.

Al apoyarse sobre el protocolo HTTP, se puede utilizar https además de no presentar problemas con cortafuegos, ya que utilizan puertos que suelen estar abiertos como el 80, 8080 o 443.

Se pueden dar tres enfoques diferentes a la hora de definir un servicio web. En ellos precisamente se basa lo que se conocen como arquitecturas orientadas a servicios (SOA – Service Oriented Architecture). Los tres enfoques corresponden a:

- Llamadas a procedimiento remotos (RPC): Se enfoca el servicio web como una colección de operaciones o procedimientos que pueden ser invocados desde una máquina diferente de donde se ejecutan.
- SOAP. Protocolo de acceso a objetos sencillos (Simple Object Access Protocol). Mientras que, en RPC la unidad básica de interacción es la operación, en SOAP, la unidad de interacción pasa a ser el mensaje. Por lo tanto, en muchos casos se conocen como servicios orientados a mensaje. Cada uno de los mensajes que se va a utilizar ha de ser definido siguiendo una estricta sintaxis expresada en XML.
- Transferencia de estado representacional (REST). En REST se utiliza directamente el protocolo HTTP, por medio de sus operaciones GET, POST, PUT y DELETE. En consecuencia, esta arquitectura se centra en la solicitud de recursos, en lugar de las operaciones o los mensajes de las alternativas anteriores.

Servicios web basados en REST

Este planteamiento supone seguir los principios de la aplicación WWW, pero, en lugar de solicitar páginas web, se solicitan servicios web.

Los principios básicos de REST son:

- Transporte de datos mediante HTTP, utilizando las operaciones de este protocolo (GET, POST, PUT y DELETE).
- Los diferentes servicios son invocados mediante una URI que identifica un recurso en Internet.
- La codificación de datos es identificada mediante tipos MIME (text/html, image/gif, etc.)

Las ventajas más significativas del uso de REST son:

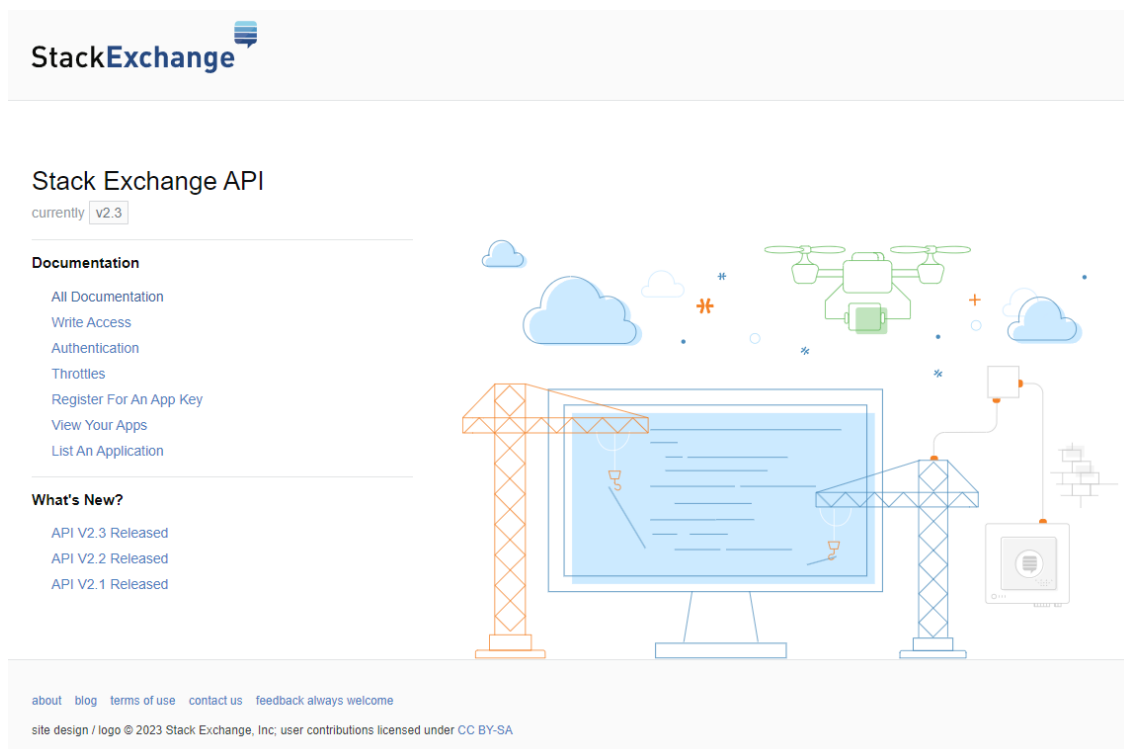
- Mejores tiempos de respuesta y disminución de sobrecarga, tanto en el cliente como en el servidor.
- Mayor estabilidad frente a futuros cambios.
- Sencillez en el desarrollo de clientes, que sólo han de ser capaces de realizar interacciones HTTP y codificar información en sistemas tipo XML/JSON.

Ejemplo de llamada a servicio web RESTFUL

En la web ^o se pueden encontrar multitud de servicios en su gran mayoría de este tipo. Muchos de ellos requieren un token o registro previo. Entre los que tienen un acceso abierto, está StackExchange que se define como una red de más de 130 comunidades Q&A (preguntas y respuestas) entre las que está Stack Overflow.

Entrando en su apartado en AnyApi, además de la descripción de los distintas llamadas a servicios, dentro de Read More, se proporciona el enlace que lleva a la web oficial de la API:

<https://api.stackexchange.com/>



13. Web de la API StackExchange

En el apartado [All Documentation](#) se pueden consultar los métodos Per-Site y para este ejemplo se elegirá [search](#) ya que permite realizar búsquedas con una palabra clave sobre el sitio Stack Overflow. El propio sitio proporciona la posibilidad de probar el servicio, lo que va a permitir obtener un enlace listo para integrarlo en el componente que se va a diseñar. En este ejemplo, se va a realizar una búsqueda con los siguientes parámetros:

- Número de página (page): 1
- Tamaño de página (pagesize): 10
- Fecha de inicio (fromdate): Un mes antes de la fecha actual
- Fecha de fin (todate): La fecha actual
- Orden (order): Descendente
- Criterio de ordenación (sort): Fecha de creación del post.
- Etiquetado (tagged): Etiqueta del post. Parámetro a introducir por el usuario.

En la siguiente imagen se puede ver un ejemplo de llamada:

Try It

Stack Overflow [\[edit\]](#)

[link](#) | [default filter](#) [\[edit\]](#) ▼

page	<input type="text" value="1"/>	pagesize	<input type="text" value="10"/>	fromdate	<input type="text" value="2022-10-31"/>
todate	<input type="text" value="2023-10-31"/>	order	<input type="text" value="desc"/>	min	<input type="text"/>
max	<input type="text"/>	sort	<input type="text" value="creation"/>	tagged	<input type="text" value="swing"/>
nottagged	<input type="text"/>	intitle	<input type="text"/>		

/2.3/search?page=1&pagesize=10&fromdate=1667174400&todate=1698710400&order=desc&sort=creation&tagged=swing&site=stackoverflow

Run

```
{
  "items": [
    {
      "tags": [
        "java",
        "swing"
      ],
      "owner": {
        "account_id": 29789543,
```

14. Llamada al servicio REST desde la web de la API

Como puede verse, el sistema proporciona el vínculo de llamada con estos parámetros. Esta API tiene una particularidad, y es que, al ser información completamente pública, no requiere ningún mecanismo de autenticación (generalmente un token) para poder hacer la llamada.

Si se abre el vínculo en el navegador, se puede consultar la salida en forma del lenguaje de marcado JSON y ver su formato para poder elaborar el componente.

```
https://api.stackexchange.com/2.3/search?page=1&pagesize=10&fromdate=1667174400&todate=1698710400&order=desc&sort=creation&tagged=swing&site=stackoverflow

1 {
2   "items": [
3     {
4       "tags": [
5         "java",
6         "swing"
7       ],
8       "owner": {
9         "account_id": 29789543,
10        "reputation": 1,
11        "user_id": 22830079,
12        "user_type": "registered",
13        "profile_image": "https://lh3.googleusercontent.com/a/ACg8ocIFsFZ3MqAYNqAzjdfqTv8KbD7UxNwHcnd7M_gS227N=k-s256",
14        "display_name": "Klaus Herz",
15        "link": "https://stackoverflow.com/users/22830079/klaus-herz"
16      },
17      "is_answered": false,
18      "view_count": 20,
19      "answer_count": 0,
20      "score": 0,
21      "last_activity_date": 1698699374,
22      "creation_date": 1698699374,
23      "question_id": 77391796,
24      "content_license": "CC BY-SA 4.0",
25      "link": "https://stackoverflow.com/questions/77391796/java-swing-text-disappearing-after-paintinng",
26      "title": "Java swing text disappearing after paintinng"
27    },
28  ],
29   "tags": [
30     "java",
31     "swing",
32     "header",
33     "jtable",
34     "jtableheader"
35   ],
36   "owner": {
37     "account_id": 19808077,
38     "reputation": 1,
39     "user_id": 14507604,
40     "user_type": "registered",
41     "profile_image": "https://www.gravatar.com/avatar/bb9f716df2393cb77325756f3948a20a?s=256&d=identicon&r=P&f=y&so-version=2",
42     "display_name": "Akilesh Balireddi",
43     "link": "https://stackoverflow.com/users/14507604/akilesh-balireddi"
44   },
45   "is_answered": false,
46   "view_count": 13,
47   "answer_count": 0,
48   "score": 0,
49   "last_activity_date": 1698673771,
50   "creation_date": 1698671284,
51   "last_edit_date": 1698673771,
52   "question_id": 77388913,
53   "content_license": "CC BY-SA 4.0",
54   "link": "https://stackoverflow.com/questions/77388913/header-column-span-in-jtable-to-show-two-columns-under-one-header-jtable-java-s",
55   "title": "Header column span in jtable to show two columns under one header- jtable Java swings"
56 },
57 ]
```

15. Salida de una llamada a API REST

Con esto ya se puede comenzar el desarrollo del componente que mostrará la salida de una búsqueda por un término realizada desde la interfaz. Para este ejemplo, la información que se va a considerar útil para su visualización es:

- El identificador del post.
- La fecha de creación.
- El título
- El autor
- La URL de acceso al post.
- Si el post tiene o no respuestas.

Por tanto, lo primero que se debe hacer es crear una clase que permita agrupar estos datos para su utilización en el componente. Como nombre se le pueda dar RegistroStack y quedar finalmente de esta forma:

```
import java.util.Date;
public class RegistroStack {
    public RegistroStack(int id, Date fechaCreacion, String titulo, String autor,
String url, boolean conRespuestas) {
        this.id = id;
        this.fechaCreacion = fechaCreacion;
        this.titulo = titulo;
        this.autor = autor;
        this.url = url;
        this.conRespuestas = conRespuestas;
    }
}
```

```
private int id;
private Date fechaCreacion;
private String titulo;
private String autor;
private boolean conRespuestas;
private String url;
public String getUrl() {
    return url;
}
public void setUrl(String url) {
    this.url = url;
}
public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public Date getFechaCreacion() {
    return fechaCreacion;
}
public void setFechaCreacion(Date fechaCreacion) {
    this.fechaCreacion = fechaCreacion;
}
public String getTitulo() {
    return titulo;
}
public void setTitulo(String titulo) {
    this.titulo = titulo;
}
public String getAutor() {
    return autor;
}
public void setAutor(String autor) {
    this.autor = autor;
}
public boolean isConRespuestas() {
    return conRespuestas;
}
public void setConRespuestas(boolean conRespuestas) {
    this.conRespuestas = conRespuestas;
}
}
```

Como puede verse, el código ya incluye los getter/setter de cada atributo.

El componente que va a permitir la búsqueda y mostrar los resultados puede tener una forma similar a la de la figura:

The screenshot shows a web application interface. At the top, there is a search bar with a text input field and a button labeled 'Buscar'. Below the search bar is a table with five columns: 'Identificador', 'Fecha de creación', 'Título', 'Autor', and 'Con respuestas'. The 'Con respuestas' column contains four empty checkboxes. At the bottom of the interface, there is a red error message that reads 'Error no controlado'.

16. Componente StackOverflowSearch

Es una interfaz simple que contiene un cuadro de texto de búsqueda (txtBusqueda), un botón (btnBuscar), una JTable (tblResultados) y una etiqueta lblError. Aunque en este ejemplo se muestra un TableModel diseñado para la ocasión, no es necesario introducir ninguno ya que el sistema lo asignará dinámicamente. Es aquí donde principalmente se introducirá la lógica de llamada y procesamiento de la información del servicio REST. Se le puede dar como nombre StackOverflowSearch. De primeras, ya se ve que la JTable necesita un TableModel personalizado para la información que se desea visualizar. Por tanto, y una vez diseñada la interfaz del componente, el siguiente paso es crear dicho TableModel. Una vez más, se opta por extender AbstractTableModel sobrescribiendo aquellos métodos que sean necesarios para personalizar la nueva clase. Por tanto, se crea una clase StackTableModel con la siguiente codificación:

```
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.List;
import javax.swing.table.AbstractTableModel;

public class StackTableModel extends AbstractTableModel {
    List<RegistroStack> lstRegistros;
    ArrayList<String> columnNames;
    ArrayList<Class> columnClasses;

    public StackTableModel(List<RegistroStack> source) {
        this.lstRegistros = source;
        columnNames = new ArrayList<>();
        columnClasses = new ArrayList<>();
        columnNames.add("Identificador");
        columnNames.add("Fecha de creación");
        columnNames.add("Título");
        columnNames.add("Autor");
    }
}
```

```
        columnNames.add("Con respuestas");
        columnClasses.add(Integer.class);
        columnClasses.add(String.class);
        columnClasses.add(String.class);
        columnClasses.add(String.class);
        columnClasses.add(Boolean.class);
    }
    @Override
    public int getRowCount() {
        return lstRegistros.size();
    }
    @Override
    public int getColumnCount() {
        return columnClasses.size();
    }
    @Override
    public Object getValueAt(int rowIndex, int columnIndex) {
        RegistroStack current = lstRegistros.get(rowIndex);
        if (current != null
            && columnIndex >= 0
            && columnIndex < getColumnCount()) {
            switch (columnIndex) {
                case 0:
                    return current.getId();
                case 1:
                    SimpleDateFormat fmt = new SimpleDateFormat("dd-MM-yyyy
HH:mm:ss");
                    return fmt.format(current.getFechaCreacion());
                case 2:
                    return current.getTitulo();
                case 3:
                    return current.getAutor();
                case 4:
                    return current.isConRespuestas();
                default:
                    return "";
            }
        }
        else
            return "";
    }
    @Override
    public Class<?> getColumnClass(int columnIndex) {
        return columnClasses.get(columnIndex);
    }
    @Override
    public String getColumnName(int column) {
        return columnNames.get(column);
    }
}
```

```
}  
}
```

En primer lugar, se declaran tres variables necesarias para trabajar con la clase: la lista de registros de StackOverFlow (RegistroStack), una lista de nombres de columnas y otro de sus clases. En el constructor se recoge una lista de objetos de tipo RegistroStack que será la que defina el contenido del TableModel. Por otra parte, se construye ad-hoc la colección de columnas dando sus nombres y tipos.

Como ya se vio en la unidad 1, se sobrescriben métodos propios de AbstractTableModel como getRowCount, getColumnCount y otros. El más interesante en este caso es getValueAt(rowIndex, columnIndex). Básicamente, primero obtiene el elemento de tipo RegistroStack que está en el índice dado por rowIndex. A continuación, se evalúa si el elemento existe y si se ha introducido un orden de columna dado. De no ser así, no se lanza excepción alguna, pero se devuelve una cadena vacía. En función del índice de columna, se devuelve el valor de su propiedad correspondiente. Es interesante pararse en la de la fecha ya que interesa que esta se muestre con su hora (es normal que haya varios posts el mismo día). Para ello se crea un objeto de tipo SimpleDateFormat con el formato requerido y se aplica sobre la fecha de creación.

Ahora toca ver cómo se puede llamar al servicio desde Java, obtener la información y trasladarla a una lista de objetos RegistroStack. Lo primero es inicializar dicha lista dentro del panel StackOverflowSearch:

```
ArrayList<RegistroStack> lstRegistros = new ArrayList<>();
```

Para llamar al servicio REST se va a utilizar la biblioteca [Unirest](#) de Mashape. Para ello, como en otras ocasiones, se necesita agregar una referencia a ella. En el repositorio de Maven a la fecha de este documento, se encuentra como última versión la 1.4.9. Por tanto, en la sección **dependencies** del archivo pom.xml hay que introducir las siguientes líneas:

```
<dependency>  
  <groupId>com.mashape.unirest</groupId>  
  <artifactId>unirest-java</artifactId>  
  <version>1.4.9</version>  
  <type>jar</type>  
</dependency>
```

Esta biblioteca permite trabajar con peticiones REST y el código obtenido en JSON. Hay que incluir una serie de imports teniendo cuidado de no escoger los incorrectos ya que hay objetos que pueden encontrarse en más de una biblioteca. Se deben incluir los siguientes en el panel StackOverflowSearch:

```
import com.mashape.unirest.http.JsonNode;  
import com.mashape.unirest.http.Unirest;  
import com.mashape.unirest.http.HttpResponse;  
import com.mashape.unirest.http.exceptions.UnirestException;  
import org.json.JSONArray;  
import org.json.JSONObject;
```

Dentro del mismo componente se va a introducir el siguiente código:

```
private void serviceSearch(String text) throws UnirestException {
    lstRegistros.clear();
    long toDate = System.currentTimeMillis() / 1000;
    long diferencia = 31560000L;
    long fromDate = toDate - diferencia;
    HttpResponse<JsonNode> jsonResponse =
Unirest.get("https://api.stackexchange.com/2.3/search")
    .header("accept", "application/json").queryString("page", "1")
    .queryString("pagesize", "10")
    .queryString("fromdate", fromDate)
    .queryString("todate", toDate)
    .queryString("order", "desc")
    .queryString("sort", "creation")
    .queryString("tagged", text)
    .queryString("site", "stackoverflow")
    .asJson();
    JSONObject jn = jsonResponse.getBody().getObject();
    JSONArray itemsArray = jn.getJSONArray("items");
    for (int x=0; x<itemsArray.length(); x++)
    {
        JSONObject currentItem = itemsArray.getJSONObject(x);
        JSONObject owner = currentItem.getJSONObject("owner");
        RegistroStack currentRegStack = new
RegistroStack(currentItem.getInt("question_id"),
                new Date(currentItem.getLong("creation_date")*1000),
                currentItem.getString("title"),
                owner.getString("display_name"),
                currentItem.getString("link"),
                currentItem.getBoolean("is_answered")
            );
        lstRegistros.add(currentRegStack);
    }
    StackTableModel tbModel = new StackTableModel(lstRegistros);
    this.tblResultados.setModel(tbModel);
}
```

A continuación, se tratan los aspectos más interesantes del código que se deben tener en cuenta:

- El servicio admite las fechas en segundos desde 1-1-1970. Por tanto, toDate, que representa la fecha tope de la consulta, se obtiene teniendo en cuenta la actual en milisegundos y dividiéndola por 1000.
- La diferencia de un año también se representa en segundos y como tipo Long.
- La fecha de inicio (fromDate) es la final restándole la diferencia.
- La petición se hace mediante Unirest.get proporcionando su [URL](#), el encabezado el cual determina el formato de salida y, a continuación, cada uno de sus parámetros en forma de queryString encadenados para cerrar con un .asJson() lo que finalmente devuelve un objeto de tipo HttpResponse sobre nodos Json (JsonNode).

- Viendo el JSON de salida, puede deducirse cómo conseguir cada uno de los datos deseados. El objeto principal de la salida de la petición es **jn** obtenido del cuerpo de la petición.
- El objeto principal contiene un array de ítems identificado como **itemsArray** obtenido con el método `getJSONArray` el cual se va a recorrer con un bucle `for`. Cada elemento se asigna a `currentItem` (llamando a `getJSONObject`) y de ahí se obtienen de forma directa datos como el identificador de la pregunta, título, fecha de creación (reconvertida a fecha en un proceso inverso al de `toDate` y `fromDate`), URL y si tiene o no respuestas.
- Para obtener el nombre del propietario del post, hay que recurrir a un elemento complejo introducido en ítems y que se denomina **owner**. Su obtención se hace mediante el método `getJSONObject` y dentro de él hay otro elemento **name** que corresponde al nombre del usuario.
- Como puede verse, hay métodos personalizados para cada tipo de datos (`getString`, `getLong`, `getInt`, `getBoolean`) que ya proporcionan el dato tipado.
- Una vez creado el `currentStack` mediante su constructor, se añade a la lista que servirá para alimentar al `StackTableModel`.

El siguiente paso consiste en programar el manejador de evento del botón de búsqueda (`btnBuscar`):

```
private void btnBuscarActionPerformed(java.awt.event.ActionEvent evt)
{
    if (txtBusqueda.getText().isBlank()){
        lblError.setVisible(true);
        lblError.setText("Por favor, introduzca un patrón de búsqueda");
    }
    else if (txtBusqueda.getText().contains(" "))
    {
        lblError.setVisible(true);
        lblError.setText("Por favor, introduzca únicamente una palabra");
    }
    else{
        try {
            serviceSearch(txtBusqueda.getText());
            lblError.setVisible(false);
        } catch (UnirestException ex) {
            Logger.getLogger(StackOverflowSearch.class.getName()).log(Level.SEVERE,
            null, ex);
            lblError.setText("Error en petición al servicio: " + ex.getMessage());
        }
    }
}
```

En el interfaz de usuario se ha incluido una etiqueta que va a mostrar si sucede algún error en la búsqueda, si se ha dejado el cuadro en blanco o si se busca más de una palabra.

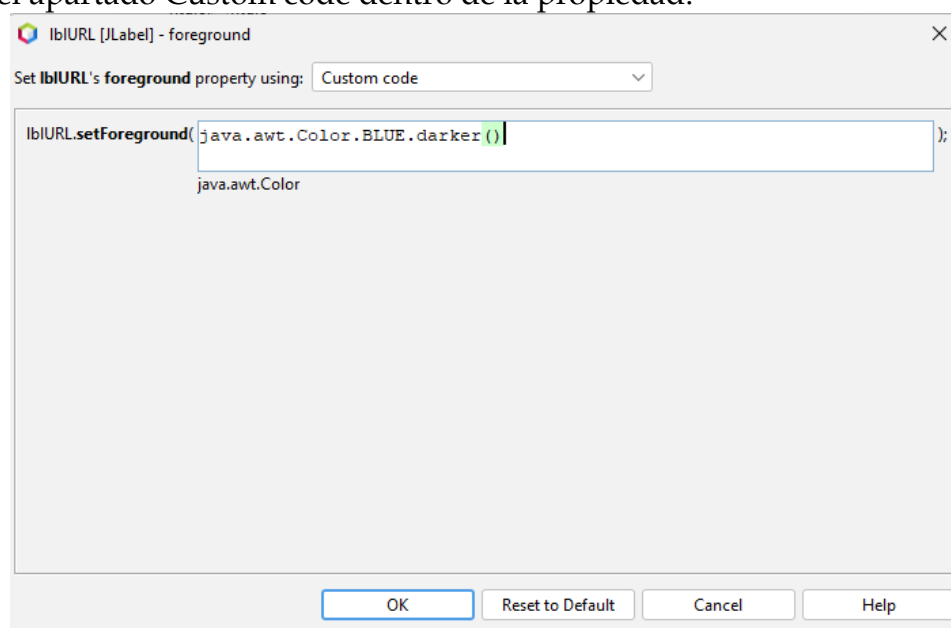
En cualquier caso, la visualización de los datos en una tabla puede ser incómoda y poco detallada, con lo que sería una buena opción que se pudiera abrir un diálogo que mostrara la información extendida incluyendo la URL del post. Para ello se utilizará un diálogo Swing añadiéndolo al proyecto mediante el menú contextual New > JDialog Form; se le puede dar como nombreDlgRegistroStack y lucir el siguiente aspecto:



17. Diálogo de consulta de información detallada

Las etiquetas colocadas a la derecha tendrán nombres descriptivos ya que son a las que se les va a dar el texto propio del registro consultado. Ejemplos: lblIdentificador, lblFechaCreacion, lblTitulo, lblAutor.

La etiqueta correspondiente a la URL (lblURL) va a tener dos características que la harán diferente del resto. Por un lado, ya que se quiere identificar que es una URL, su texto debe ir de otro color. El color se cambia en la propiedad **foreground** y el más adecuado para este caso es **BLUE.darker**. Para ajustarlo, ya que llevará un código personalizado, se puede introducir como tal en el apartado Custom code dentro de la propiedad:



18. Cambio de color de etiqueta URL

Otra propiedad que se puede cambiar es la del puntero al posarse sobre el control. Como lo normal es que, al acceder a un enlace el puntero se cambie por una mano apuntando, hay que modificar la propiedad Cursor por "Hand Cursor". Por último, hay que programar un evento de clic de ratón sobre el control para que se abra automáticamente el enlace:

```
private void lblURLMouseClicked(java.awt.event.MouseEvent evt)
{
    try {
        Desktop.getDesktop().browse(new URI(lblURL.getText()));
    } catch (IOException | java.net.URISyntaxException e1) {
        e1.printStackTrace();
    }
}
```

Para que el diálogo se abra con el contenido de un RegistroStack, hay que añadir un constructor que permita crear un diálogo con esos datos. Lo primero que hay que hacer es diseñar un método que ajuste los controles con los datos de un **RegistroStack** "rs" que será privado en la clase DlgRegistroStack:

```
private RegistroStack regStack;
private void setData() {
    this.lblIdentificador.setText(this.regStack != null ?
String.valueOf(this.regStack.getId()) : "");
    this.lblAutor.setText(this.regStack != null ? this.regStack.getAutor() : "");
    SimpleDateFormat fmt = new SimpleDateFormat("dd-MM-yyyy HH:mm:ss");
    this.lblFecha.setText(this.regStack != null ?
fmt.format(regStack.getFechaCreacion()) : "");
    this.lblTitulo.setText(this.regStack != null ? this.regStack.getTitulo() : "");
    this.lblURL.setText(this.regStack != null ? this.regStack.getUrl() : "");
    this.chkConRespuestas.setSelected(this.regStack != null ?
regStack.isConRespuestas() : false);
}
```

El método se explica solo y, como puede verse, incluye la visualización de la fecha con un formato concreto y también contempla la posibilidad de un RegistroStack nulo. El constructor quedaría así:

```
public DlgRegistroStack(java.awt.Frame parent, boolean modal, RegistroStack rs) {
    super(parent, modal);
    initComponents();
    this.regStack = rs;
    setData();
}
```

Importante: El diálogo **incluye un método main que hay que eliminar**, ya que usa el constructor por defecto y no es compatible con el que se acaba de incluir. Dado que no se va a usar, no es necesario en este proyecto.

Por último, es interesante incluir un método para destruir el diálogo cuando se cierre:

```
private void formWindowClosed(java.awt.event.WindowEvent evt)
{
    this.dispose();
}
```

Una vez diseñado por completo el diálogo y codificado todo lo necesario, el último paso para el control consiste en programar un evento de doble clic sobre un elemento de la tabla. Para ello hay que volver al panel StackOverflowSearch y programar el manejador de evento sobre MousePressed. En este, se recoge la fila que está ubicada donde apunta el cursor del ratón y si se hace doble clic sobre ella abriéndose el nuevo diálogo con el RegistroStack seleccionado.

```
private void tblResultadosMousePressed(java.awt.event.MouseEvent evt)
{
    JTable table = (JTable) evt.getSource();
    Point point = evt.getPoint();
    int row = this.tblResultados.rowAtPoint(point);
    if (evt.getClickCount() == 2 && table.getSelectedRow() != -1) {
        RegistroStack rs = this.lstRegistros.get(row);
        DlgRegistroStack dlgRs = new DlgRegistroStack((JFrame)
this.getTopLevelAncestor(), true, rs);
        dlgRs.setVisible(true);
    }
}
```

Es interesante la codificación de este manejador ya que se puede ver cómo recoger el control origen del evento (con `evt.getSource()`), el punto en el que se produce (`getPoint()`) y el índice de fila al que apunta (`rowAtPoint()`). En realidad, esto no es estrictamente necesario ya que se puede obtener el mismo dato desde el método `getSelectedRow()`. Dado que el `TableModel` recoge filas de tipo `RegistroStack` de una lista subyacente, basta con elegir el índice correspondiente a dicha fila de la lista y enviárselo al diálogo. Al haber utilizado un panel como componente, no se le puede enviar al diálogo como padre ya que no es un tipo admitido. Se usa entonces un `getTopLevelAncestor()` para obtener el formulario padre del control y se le envía al diálogo junto con el booleano que indica si es modal y, por último, el `RegistroStack` correspondiente.

Conclusiones

Este último caso está encaminado principalmente a ilustrar el consumo de un servicio RESTFUL, pero, además contiene aportaciones interesantes sobre el manejo de `JTables` (algunas ya vistas), diálogos, eventos con comportamientos no estándar, etc.

ÍNDICE DE FIGURAS

1. Panel para validación tradicional.....	3
2. Asociación de tipo a JComboBox	7
3. Propiedad de un componente	10
4. Edición de propiedades de tipo de letra.	12
5. Panel para recoger los colores del componente	15
6. Vista de diseño del BeanInfo del componente.	18
7. Propiedad ajustada desde editor.....	18
8. Inserción de código automatizada	19
9. Editor de propiedades de imagen y opacidad	22
10. JavaBean con PropertyEditorSupport	25
11. Editor de propiedades para imagen de fondo.	25
12. Empaquetado de componentes	37
13. Web de la API StackExchange	40
14. Llamada al servicio REST desde la web de la API	41
15. Salida de una llamada a API REST	42
16. Componente StackOverFlowSearch	44
17. Diálogo de consulta de información detallada	49
18. Cambio de color de etiqueta URL	49

BIBLIOGRAFÍA - WEBGRAFÍA

- González, P. (2016) *UT3. Creación de componentes visuales (Java Swing)*.
<https://youtube.com/playlist?list=PLIfP1vJ2qaklWieJgK6uo0ppg-itYlXxV&si=ktZEDD0fcf5VJNyY>
- Groussard, T. et al. (2020). *Java 11. Los fundamentos del lenguaje Java*. Editorial Eni.
- García Miguel, D. et al. (2021). *Desarrollo de interfaces*. Editorial Síntesis.
- Álvarez Caudales, C. (2021) *Java Record Class y JDK 14*
<https://www.arquitecturajava.com/java-record-class-y-jdk-14/>
- Evans, B. (2023) *Reflection for the modern Java programmer*
<https://blogs.oracle.com/javamagazine/post/java-reflection-introduction>