USOS INTERESANTES DE JFormattedTextField

Francesc Rosés i Albiol
(08/2002)

La versión 1.4 del JDK de Sun nos obsequia con un montón de novedades, todas ellas buenas y algunas, largamente deseadas.

En este artículo nos centraremos en algunos aspectos interesantes y poco comentados de un nuevo componente Swing: `JFormattedTextField`.

Este componente ha sido muy esperado por todos a los que nos ha tocado desarrollar interficies de usuario usando Java. Lo primero que uno piensa cuando se acerca por primera vez a él es "*finalmente tenemos un campo que soporta máscaras"*. Todos esperábamos este componente. Sin embargo, su nombre no es algo así como *JMaskedTextField*. Su nombre nos insinúa que va más allá de un campo con máscaras. Se trata de un campo de entrada que nos permite especificar formatos. Ciertamente, una manera de especificar un formato de entrada es mediante una máscara, pero no es el único. Además, no hay que olvidar que un campo de entrada no sólo sirve para entrar texto. También muestra lo que hemos entrado. Volveremos más tarde sobre este punto.

Resumiendo un poco las características de este componente podemos decir que:

- Permite especificar el formato de entrada de datos mediante máscaras y sabe aprovechar el resto de especificaciones de formato disponibles en Java para números, fechas, horas, etc.
- Permite decidir si se admiten caracteres incorrectos en la entrada o no.
- Permite distinguir entre modalidad de edición y modalidad de visualización.
- Permite que decidamos qué hacer con el foco si lo que el usuario ha entrado no es correcto.

Vamos a desarrollar un poco estos puntos.

Especificación de formato

El componente `JFormattedTextField` nos permite especificar el formato de diversas maneras:

De manera automática: Asignando un valor al campo

Simplemente assignando un valor al campo, éste nos asigna un formato. Así, por ejemplo, si le asignamos una fecha, él nos la permitirá editar. El siguiente código crea un campo de entrada para fechas con el formato por defecto:

```
JFormattedTextField efFecha = new JformattedTextField(new Date());
```

El campo mostrará la fecha actual con el siguiente formato:

```
19-ago-2002
```

Pero no sólo nos presenta la fecha. Nos permite editarla de una manera sencilla y sin error posible. Si colocamos, por ejemplo, el cursor sobre el mes y pulsamos la flecha hacia arriba, el mes cambia y pasa a ser *sept*. Si pulsamos la flecha hacia abajo, el mes será *jul*. El mismo comportamiento se da para el día y el año. Además, el comportamiento es inteligente. Supongamos que la fecha sea 28 de febrero de 2002 y que aumentemos el día. La nueva fecha sería 1 de marzo de 2002. Lógicamente, si el año fuera el 2000 (bisiesto) la fecha propuesta sería el 29 de febrero de 2000.

Mediante una máscara

Podemos utilizar una máscara para determinar el formato. Por ejemplo, si quisiéramos crear un campo para entrar códigos de cuenta corriente, podríamos hacerlo así de fácil:

```
MaskFormatter mfCC = new MaskFormatter("####-####-##-##########");
mfCC.setPlaceholderCharacter('_');
JFormattedTextField efCC = new JformattedTextField(mfCC);
```

El campo tendría el siguiente aspecto:

```
____-____-__-__________
```

Fijémonos que las partes escribibles se representan con el carácter de subrayado que hemos especificado con `setPlaceholderCharacter()`.

La siguiente tabla resume los caracteres utilizables en una máscara:

Carácter	Descripción
#	Un número
?	Una letra
A	Una letra o un número
*	Cualquier cosa
U	Una letra que será pasada a mayúscula
L	Una letra que será pasada a minúscula
H	Un dígito hexadecimal (A-F, a-f, 0-9)
'	Carácter de escape para otro caràcter de máscara

Mediante descriptores de formato ya existentes

Java nos ofrece una amplia gama de especificaciones de formato para fechas, horas, números y monedas. Todos ellos pueden ser utilizados, directa o indirectamente, para especificar el formato usado por el campo. Ejemplificaremos algunos de ellos.

Más arriba, hemos mostrado cómo especificar un formato de fecha simplemente pasando una al constructor del campo. El resultado es vistoso, pero cualquier persona que entre datos nos dirá que es poco práctico. Sería más interesante algo del estilo de *dd/mm/aa*. El siguiente código nos muestra cómo hacerlo:

```
JFormattedTextField efFecha =
  new JFormattedTextField(new SimpleDateFormat("dd/MM/yy"));
```

El resultado obtenido sería: 19/08/02. El comportamiento de las flechas sería el ya descrito.

Si lo que pretendemos es entrar un número con un máximo de dos decimales:

```
JFormattedTextField efNum =
  new JformattedTextField(new DecimalFormat("#,###.00"));
```

Si nos interesa que el usuario entre importes en euros en nuestro campo:

```
JFormattedTextField efMon =
  new JformattedTextField(NumberFormat.getCurrencyInstance());
efMon.setValue(new Integer(1000));
```

Lo que veríamos sería:

```
1.000,00 €
```

Admitir o no caracteres incorrectos

A veces, nos puede interesar permitir que el usuario pueda entrar caracteres incorrectos. Para hacerlo, usaremos formateadores propios de *JFormattedTextField*. Veamos el siguiente ejemplo:

```
JFormattedTextField efNum =
  new JformattedTextField(new DecimalFormat("#,###.00"));
NumberFormatter nf = (NumberFormatter)efNum.getFormatter();
nf.setAllowsInvalid(true);
```

Disponemos de tres formateadores especiales derivados todos ellos de la clase *javax.swing.JFormattedTextField.AbstractFormatter*:

- MaskFormatter – Utilizado para máscaras y derivado directamente de *AbstractFormatter*.
- NumberFormatter – Utilizado para números y derivado de una subclase de *AbstractFormatter*: *InternationalFormatter*.
- DateFormatter – Utilizado para fechas y horas y derivado, como el anterior, de *InternationalFormatter*.

Insertar o sobrescribir

Algo de sumo interés es poder especificar si insertamos o sobrescribimos caracteres. Lo ideal sería que se pudiera decidir pulsando la tecla *Ins*, pero esto no es inmediato.

El siguiente ejemplo nos indica cómo permitir la sobrescritura:

```
JFormattedTextField efNum =
  new JformattedTextField(new DecimalFormat("#,###.00"));
NumberFormatter nf = (NumberFormatter)efNum.getFormatter();
nf.setOverrideMode(true);
```

Modalidad de edición y de visualización

Imaginemos que tenemos un campo para entrar importes en euros como el que hemos descrito más arriba. El resultado obtenido es interesante; se ve bien, con el símbolo de euro al final y los separadores de milllares y la coma decimal siguiendo las directrices de nuestro país. Pero una vez más, un usuario que se dedique a entrar datos nos diría que es incómodo. Normalmente, se usa el teclado numérico y uno no tiene que ir a buscar la coma al teclado alfanumérico. Usa un punto para indicar la coma. Claro que si bien es práctico entrar *1234.35*, queda mal cuando se visualiza. Tenemos, pues, un conflicto: lo que es práctico para la entrada no es claro en la visualización.

JFormattedTextField nos permite resolver este conflicto especificando un formato para la edición y otro para la visualización. Cuando el foco esté en el campo, usará el de edición y cuando éste pierda el foco, usará el de visualización.

Veamos cómo hecerlo para nuestro campo de importes en euros:

```
// Creamos el campo
JFormattedTextField efDecimal =
   new JformattedTextField();

// Formato de visualización
NumberFormat dispFormat = NumberFormat.getCurrencyInstance();

// Formato de edición: inglés (separador decimal: el punto)
NumberFormat editFormat =
               NumberFormat.getNumberInstance(Locale.ENGLISH);

// Para la edición, no queremos separadores de millares
editFormat.setGroupingUsed(false);
```

```
// Creamos los formateadores de números
NumberFormatter dnFormat = new NumberFormatter(dispFormat);
NumberFormatter enFormat = new NumberFormatter(editFormat);

// Creamos la factoría de formateadores especificando los
// formateadores por defecto, de visualización y de edición
DefaultFormatterFactory currFactory =
  new DefaultFormatterFactory(dnFormat, dnFormat, enFormat);

// El formateador de edición admite caracteres incorrectos
enFormat.setAllowsInvalid(true);

// Asignamos la factoría al campo
efDecimal.setFormatterFactory(currFactory);
```

Editamos en formato inglés (usamos el punto como separador decimal) y sin separadores de millares.

Visualizamos lo que hemos entrado en el formato de moneda y numérico de nuestro país. En este caso, el euro como símbolo de moneda, el punto como separador de millares y la coma como separador decimal, pero si el programa se ejecutara en Inglaterra, verían el símbolo de la Libra, la coma sería el separador de millares y el punto, el separador decimal.

La siguiente imagen muestra nuestro campo en modalidad de edición:

La siguiente imagen muestra el mismo campo en modalidad de visualización:

Si observamos el código, veremos que opto por admitir caracteres incorrectos en edición. La elección no es gratuita ni responde, como debiera, a cuestiones de ergonomía. Si no los admitiera, no podría teclear el punto en modalidad de edición, a no ser que lo hiciera entre dos dígitos. Esto es, no podría escribir "1234." pero sí "123435" y a continuación poner el punto entre el 4 y el 3. A falta de más minuciosas comprobaciones, mi impresión es que hay un *bug* más que notable.

Control del foco

Uno de los problemas típicos en el desarrollo de interficies gráficas para la entrada de datos es decidir qué se hace cuando el usuario que ha rellenado incorrectamente un campo quiere pasar al siguiente. En algunos casos nos puede interesar que lo pueda hacer, pero en otros no.

Imaginemos que necesitamos un campo para entrar números de DNI, con su letra. Este campo, en nuestra aplicación, es clave ya que a partir de él, se obtiene el resto de la información. Así, pues, si el número entrado fuera incorrecto, no debiéramos permitir que el usuario saltara al campo siguiente.

Anteriormente, había que utilizar algún truco (que no viene al caso) para evitar que el foco se fuera al siguiente componente. En la versión 1.4, se nos facilita bastante el trabajo. Podemos asignar al campo una clase que extienda `InputVerifier` mediante el método *`setInputVerifier()`* de manera que sea esta clase la que controle si el usuario podrá salir del campo (pasar el foco a otro componente) o no.

Para seguir con el ejemplo del DNI, proponemos un pequeño programa que ilustre el procedimiento. Veamos primero algunas partes relevantes del programa el listado completo está disponible en el

apéndice A.

Vamos a definir una máscara que facilite la entrada de DNIs en nuestro campo:

```
// Definición de la máscara de DNI
MaskFormatter maskDNI = null;
try {
  maskDNI = new MaskFormatter("##.###.###-U");
} catch (ParseException e) {
    // De momento, no hacemos nada
  }
// El caràcter a mostrar en las posiciones escribibles es el
// subrayado.
maskDNI.setPlaceholderCharacter('_');
```

La máscara obligará al usuario a entrar ocho dígitos y una letra que será pasada a mayúsculas. Además, mediante el método `setPlaceholderCharacter()`, asignamos un carácter de subrayado para que sirva de pauta al usuario, indicándole las posiciones editables del campo.

El carácter *U* que vemos en la máscara obligará al usuario a escribir la letra del DNI y pasará dicha letra a mayúsculas.

La máscara se encargará, pues, de que el usuario escriba dígitos y letras donde corresponda, pero el valor entrado no será entregado al campo directamente hasta que pulsemos la tecla `Intro`. Al cambiar de foco, el `MaskFormatter` no entrega el valor. Hay que decirle explícitamente que si lo editado es válido, pase el valor al campo. Para ello, utilizaremos el método `setCommitsOnValidEdit(boolean)`.

```
maskDNI.setCommitsOnValidEdit(true);
```

Si comentamos esta línea, veremos que al entrar un DNI incorrecto nos deja cambiar el foco debido a que el valor no se ha entregado al campo para que determine si debe permitir el cambio de foco o no.

Finalmente, creamos el campo:

```
JFormattedTextField efDNI = new JformattedTextField(maskDNI);
```

En este momento, ya hemos dotado a nuestro campo de un cierto control para entrar DNIs:

- Nos fuerza a escribir los números y la letra en los lugares que corresponde
- Passa automáticamente la letra final de control a mayúsculas

Sin embargo, la máscara no nos proporciona todo el control que necesitamos. Si la persona que entra los datos se equivoca en la letra de control, el error queda registrado. Necesitamos, pues, impedir que la persona que entra los datos entre un DNI erróneo (aunque, y de eso se encarga la máscara, bien formado).

La versión 1.4 del JDK incorpora un nuevo método a la classe `java.awt.Component`: `setInputVerifier(InputVerifier v)`. Este nuevo método nos permite asignar a un `JFormattedTextField` un algoritmo de control del contenido entrado. Este algoritmo de control se hallará embebido en una subclasse de `InputVerifier`. La classe `InputVerifier` es abstracta y obliga a sus subclases a implementar el método `public boolean verify (JComponent input)`. Este método devuelve `true`, si la comprobación es correcta, o `false`, si no lo es.

Veamos ahora la classe derivada de *InputVerifier* que se encarga de verificar si el DNI entrado es correcto y permite al campo decidir si autoriza, o no, el cambio de foco.

```
class ValidateDNI extends InputVerifier {
  String pattern = "RWAGMYFPDXBNJZSQVHLCKET";
```

```
  /**
   * Verifica un DNI.
   */
  private boolean verifyDNI(String DNI) {
    // Eliminamos caracteres de separación.
    DNI = DNI.replaceAll("[.-]", "");
    if (DNI.length() != 9) {
      return false;
    }

    // El último carácter debe ser una letra
    if (!Character.isLetter(DNI.charAt(8))) {
      return false;
    }
    int digits;

    try {
      digits = Integer.parseInt(DNI.substring(0, 8));
    } catch (NumberFormatException e) {
        return false;
      }

    // El algoritmo mágico
    int pos = (digits%23);
    if (pos == 0) {
      pos = pattern.length();
    }
    pos = pos -1;   // Las tiras en Java están basadas en cero!
    return (pattern.charAt(pos) == DNI.charAt(8));
}

/**
 * Sobrescribimos el método del padre para realizar la
 * comprobación del DNI entrado.
 */
public boolean verify(JComponent input) {
  if (input instanceof JFormattedTextField) {
    Object o = ((JFormattedTextField)input).getValue();
    if (o == null) return true;

    String value = o.toString();
    return verifyDNI(value);
  }
  return false;
}
```

El método `verify()` se encarga de llamar al método `verifyDNI()` que contiene el algoritmo de verificación (simplificada) de DNIs.

CONCLUSIONES

La versión 1.4 del JDK nos aporta un nuevo componente, `JFormattedTextField`, que llena un vacío importante para los desarrolladores de interfícies de usuario, permiténdoles un control importante sobre los datos entrados por los usuarios y evitando en muchos casos errores típicos de entrada como en el caso de las fechas.

Además, `JFormattedTextField` nos permite separar fácilmente la entrada de datos de la visualización de los mismos, empleando el formato adecuado para cada una de las situaciones.

Si bien `JFormattedTextField` nos permite un control importante sobre el formato de los

datos, entendemos que este control no es suficiente ya que, en algunos casos, es necesaria una verificación de la corrección del contenido entrado que va más allá del puro formato (p.e. el caso del D.N.I. o del código de cuenta corriente). Por ello, la versión 1.4 nos proporciona mecanismos de control como la classe `InputVerifier` y el método `setInputVerifier` de la classe `Component` que proporcionan la flexibilidad necesaria para poder establecer con facilidad este orden de comprobaciones.