Manejo de ristras de caracteres usando expresiones regulares

Introducción

Las expresiones regulares proporcionan una gran potencia a la hora de expresar patrones de búsqueda de una subristra en una ristra. Java ofrece operaciones, tanto en la clase *String* como en otras más específicas, para sacar partido de esta potencia.

Operaciones de la clase *String* basadas en expresiones regulares

La clase *String* de Java ofrece cuatro métodos que usan expresiones regulares: *matches*, *replaceAll*, *replaceFirst* y *split*.

El método *matches* tiene un parámetro de la clase *String* que representa una expresión regular y devuelve un valor de tipo *boolean* que indica si la ristra sobre la que se aplica cumple, o no, el patrón representado por dicha ristra:

```
String tel = "555-329099";
String dni = "42876234M";
String pattern = "\\d{8}[A-Z]";
boolean b1 = tel.matches(pattern); // false
boolean b2 = dni.matches(pattern); // true
```

El método *replaceAll* tiene dos parámetros de la clase *String*, el primero representa una expresión regular y el segundo una ristra. El método devuelve un nuevo objeto de la clase *String* cuyo valor es igual al de la ristra sobre la que se aplica, cambiando todas las subristras que cumplan el patrón representado por la expresión regular por la ristra representada por el segundo parámetro:

```
String tel = "tel. del trabajo: 555-329099, tel. de casa: 555-549349";
String pattern = "\\d{3}-\\d{6}";
String nueva = tel.replaceAll(pattern, "<tel>");
    // nueva es igual a "tel. del trabajo: <tel>, tel. de casa: <tel>"
```

El método *replaceFirst* es similar al *replaceAll*, pero sólo sustituye la primera ocurrencia:

```
String tel = "tel. del trabajo: 555-329099, tel. de casa: 555-549349";
String pattern = "\\d{3}-\\d{6}";
String nueva = tel.replaceFirst(pattern, "<tel>");
// nueva es igual a
// "tel. del trabajo: <tel>, tel. de casa: 555-549349"
```

El método *split* devuelve un *array* de objetos de la clase *String* que resulta de dividir la ristra sobre la que se aplica en subristras, usando como separador las subristras que cumplan con el patrón expresado por la expresión regular que es su único parámetro:

```
String s1 = "esto es  un   ejemplo";
String pattern = "\\s+";
String[] words = s1.split(pattern);
    // words == {"esto", "es", "un", "ejemplo"}
```

El *array* resultante podría contener ristras vacías, cuando el patrón se repita dos veces seguidas:

```
String s1 = "esto es  un   ejemplo";
String pattern = "\\s";
String[] words = s1.split(pattern);
    // words == {"esto", "es", "", "un", "", "" "ejemplo"}
```

(Nótese que la diferencia entre el último ejemplo y el penúltimo es que en el último *pattern* expresa "1 espaciador", mientras que en el penúltimo expresa "1 o más espaciadores", lo que implica que, en el último, se tome una subristra vacía entre cada dos espacios consecutivos).

Las clases *Pattern* y *Matcher*

Operaciones básicas de las clases *Pattern* y *Matcher*

Java ofrece dos clases específicamente diseñadas para manipular ristras de caracteres de forma eficiente usando expresiones regulares: *Pattern* y *Matcher*[1].

La clase *Pattern* sirve, básicamente, para definir un patrón de búsqueda mediante una expresión regular. Un objeto de la clase *Pattern* se crea mediante el método *compile*, pasando una ristra con la expresión regular del patrón a representar:

```
Pattern p = Pattern.compile("\\s");
```

El objeto *Pattern* se puede usar para dividir una ristra, usando el patrón como separador:

```
String line = "esto es un ejemplo";
String[] words = p.split(line);
    // words == {"esto", "es", "un", "ejemplo"}
```

También se puede usar para crear un objeto *Matcher* para manipular una ristra:

```
Matcher m = p.matcher(line);
```

La clase *Matcher* sirve para buscar coincidencias de un patrón, expresado mediante una expresión regular, en una ristra. Un objeto de la clase *Matcher* se crea a partir de un objeto de la clase *Pattern* y una ristra a tratar:

```
Pattern p = Pattern.compile("\\w+");
String line = "esto es un ejemplo";
Matcher m = p.matcher(line);
```

El método *find* localiza la primera, o la siguiente, aparición de una subristra que coincida con el patrón:

```
while (m.find()) …
```

[1] Para usarlas, hay que incluir las siguientes cláusulas:

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

Los métodos *start* y *end* devuelven las posiciones de inicio y fin (más 1) de la subristra encontrada en la iteración actual:

```
System.out.print(m.start());
System.out.print(", " + m.end() + ": ")
```

0, 4:

El método *group* devuelve la subristra encontrada en la última ejecución de *find*:

```
System.out.println(m.group());
```

esto

Los tres métodos, *start*, *end* y *group* pueden llevar como parámetro un número para referirse a una subristra de la subristra encontrada que corresponda con una subexpresión o grupo numerado:

```
                                |---$1---|---$2---|---$3---|
Pattern p = Pattern.compile("(\\d{2})-(\\d{2})-(\\d{4})");
String line = "10-12-2009";
Matcher m = p.matcher(line);
m.find();
String date = m.group(); // "10-12-2009"
String day = m.group(1); // "10"
String month = m.group(2); // "12"
String year = m.group(3); // "2009"
```

La subristra completa corresponde al grupo 0.

Esquema general de manipulación de ristras

Con las operaciones básicas de las clases *Pattern* y *Matcher* podemos diseñar un esquema general para la manipulación de ristras usando expresiones regulares. Este esquema consta de tres elementos:

1. Inicialización, donde se define el patrón (un objeto *Pattern*) y se crea el objeto *Matcher* que lo utilizará.
2. Iteración, mediante un bucle controlado por el método *find*, que permite ir accediendo a las sucesivas ocurrencias de subristras que cumplan el patrón.

3. Tratamiento, dentro del bucle, donde se tiene acceso a la ocurrencia actual y se puede usar para cualquier propósito que requiera el problema concreto a resolver.

Usando los ejemplos de la sección anterior, la Ilustración 1 muestra todos los elementos del esquema general para la manipulación de ristras de caracteres usando expresiones regulares que se propone.

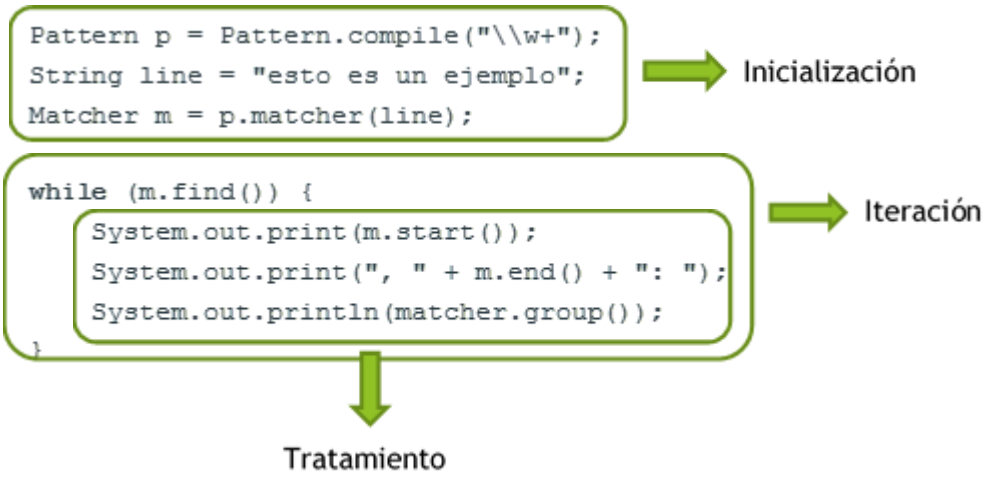

Ilustración 1

Otras operaciones de la clase Matcher

La clase *Matcher* ofrece otras operaciones interesantes, de las que destacamos:

resultado	Método
boolean	lookingAt()
	Mira si el principio de la ristra encaja con el patrón de búsqueda
boolean	matches()
	Mira si la ristra completa encaja con el patrón de búsqueda
Pattern	pattern()
	Devuelve el Pattern que se está usando
String	replaceAll (String replacement)

resultado	Método
	Devuelve una nueva String en la que se han sustituido todas las subristras que casen con el patrón por la ristra pasada como parámetro
String	replaceFirst (String replacement)
	Devuelve una nueva String en la que se ha sustituido la primera subristra que case con el patrón por la ristra pasada como parámetro

Tanto para *replaceAll* como para *replaceFirst*, la ristra de sustitución pasada como parámetro puede hacer referencia a las subristras encontradas, usando la codificación de los grupos numerados explicada anteriormente (en este caso, los grupos se introducen con el carácter '$'). El siguiente ejemplo usa esta característica para transformar una fecha que está originalmente en el formato dd-mm-aaaa:

```
                                |---$1---|---$2---|---$3---|
Pattern p = Pattern.compile("(\\d{2})-(\\d{2})-(\\d{4})");
Matcher m = p.matcher("12-02-1999");
String result = m.replaceAll("(día:$1,mes:$2,año:$3)");
// result toma el valor: "(día:12,mes:02,año:1999)"
```

Nótese que el carácter '$' en este contexto está usado como un metacarácter con un significado específico, que no es el habitual en una expresión regular. Si la ristra de sustitución tiene que incluir el carácter dólar como tal, habría que ponerlo como: "\\$", usando una barra como carácter de escape para evitar que Java interprete el $ y otra barra como carácter de escape de la propia barra.