

6.C. Expresiones regulares.

1. Expresiones regulares.

1.3. Expresiones regulares (III).

¿Te resultan difíciles las expresiones regulares? Al principio siempre lo son, pero no te preocupes. Hasta ahora has visto como las expresiones regulares permiten verificar datos de entrada, permitiendo comprobar si un dato indicado sigue el formato esperado: que un DNI tenga el formato esperado, que un email sea un email y no otra cosa, etc. Pero ahora vamos a dar una vuelta de tuerca adicional.

Los paréntesis, de los cuales no hemos hablado hasta ahora, tienen un significado especial, permiten indicar repeticiones para un conjunto de símbolos, por ejemplo: "(#{01}){2,3}". En el ejemplo anterior, la expresión "#{01}" admitiría cadenas como "#0" o "#1", pero al ponerlo entre paréntesis e indicar los contadores de repetición, lo que estamos diciendo es que la misma secuencia se tiene que repetir entre dos y tres veces, con lo que las cadenas que admitiría serían del estilo a: "#0#1" o "#0#1#0".

Pero los paréntesis tienen una función adicional, y es la de permitir definir grupos. Un grupo comienza cuando se abre un paréntesis y termina cuando se cierra el paréntesis. Los grupos permiten acceder de forma cómoda a las diferentes partes de una cadena cuando esta coincide con una expresión regular. Lo mejor es verlo con un ejemplo (seguro que te resultará familiar):

```
Pattern p=Pattern.compile("( [XY]? ) ( [0-9]{1,9} ) ( [A-Za-z] )");
```

```
Matcher m=p.matcher("X123456789Z Y00110011M 999999T");
```

```
while (m.find())
```

```
{
```

```
System.out.println("Letra inicial (opcional):"+m.group(1));
```

```
System.out.println("Número:"+m.group(2));
```

```
System.out.println("Letra NIF:"+m.group(3));
```

```
}
```

Usando los grupos, podemos obtener por separado el texto contenido en cada uno de los grupos. En el ejemplo anterior, en el patrón hay tres grupos: uno para la letra inicial (grupo 1), otro para el número del DNI o NIE (grupo 2), y otro para la letra final o letra NIF (grupo 3). Al ponerlo en grupos, usando el método `group()`, podemos extraer la información de cada grupo y usarla a nuestra conveniencia.

Ten en cuenta que el primer grupo es el 1, y no el 0. Si pones `m.group(0)` obtendrás una cadena con toda la ocurrencia o coincidencia del patrón en la cadena, es decir, obtendrás la secuencia entera de símbolos que coincide con el patrón.

En el ejemplo anterior se usa el método `find`, éste buscará una a una, cada una de las ocurrencias del patrón en la cadena. Cada vez que se invoca, busca la siguiente ocurrencia del patrón y devolverá `true` si ha encontrado una ocurrencia. Si no encuentra en una iteración ninguna ocurrencia es porque no existen más, y retornará `false`, saliendo del bucle. Esta construcción `while` es muy típica para este tipo de métodos y para las iteraciones, que veremos más adelante.

Lo último importante de las expresiones regulares que debes conocer son las secuencias de escape. Cuando en una expresión regular necesitamos especificar que lo que tiene que haber en la cadena es un paréntesis, una llave, o un corchete, tenemos que usar una secuencia de escape, dado que esos símbolos tienen un significado especial en los patrones. Para ello, simplemente antepondremos `"\"` al símbolo.

Por ejemplo, `"\"` significará que debe haber un paréntesis en la cadena y se omitirá el significado especial del paréntesis. Lo mismo ocurre con `"\"`, `"\"`, etc. Lo mismo para el significado especial del punto, éste, tiene un significado especial (¿Lo recuerdas del apartado anterior?) salvo que se ponga `"\"`, que pasará a significar "un punto" en vez de "cualquier carácter". La excepción son las comillas, que se pondrían con una sola barra: `"\"`.

Para saber más

Con lo estudiado hasta ahora, ya puedes utilizar las expresiones regulares y sacarles partido casi que al máximo. Pero las expresiones regulares son un campo muy amplio, por lo que es muy aconsejable que amplíes tus conocimientos con el siguiente enlace:

[Tutorial de expresiones regulares en Java \(en inglés\).](#)

