3️⃣

Expresiones regulares

La función de las expresiones regulares es permitir comprobar si una cadena sigue o no un patrón preestablecido. Las expresiones regulares son un mecanismo para describir esos patrones, y se construyen de una forma relativamente sencilla. Existen muchas librerías diferentes para trabajar con expresiones regulares, y casi todas siguen, más o menos, una sintaxis similar, con ligeras variaciones. Dicha sintaxis nos permite indicar el patrón de forma cómoda, como si de una cadena de texto se tratase, en la que determinados símbolos tienen un significado especial.

Reglas generales para construir una expresión regular:

- Podemos indicar que una cadena contiene un conjunto de símbolos fijo, simplemente poniendo dichos símbolos en el patrón, excepto para algunos símbolos especiales que necesitarán un carácter de escape, como veremos más adelante.
- `"[xyz]"` → Entre corchetes podemos indicar opcionalidad. Solo uno de los símbolos que hay entre los corchetes podrá aparecer en el lugar donde están los corchetes. Los corchetes representan una posición de la cadena que puede tomar uno de varios valores posibles.
- `"[a-z]" "[A-Z]" "[a-zA-Z]"` → Usando el guion y los corchetes podemos indicar que el patrón admite cualquier carácter entre la letra inicial y la final. Es importante que sepas que se diferencia entre letras mayúsculas y minúsculas, no son iguales de cara a las expresiones regulares.
- `"[0-9]"` → Y nuevamente, usando un guion, podemos indicar que se permite la presencia de un dígito numérico entre 0 y 9, cualquiera de ellos, pero solo uno.

Con las reglas anteriores podemos indicar el conjunto de símbolos que admite el patrón y el orden que deben tener. Si una cadena no contiene los símbolos especificados en el patrón, en el mismo orden, entonces la cadena no encajará con el patrón.

Indicar repeticiones:

- `"a?"` → Usaremos el interrogante para indicar que un símbolo puede aparecer una vez o ninguna. De esta forma, la letra "a" podrá aparecer una vez o simplemente no aparecer.
- `"a*"` → Usaremos el asterisco para indicar que un símbolo puede aparecer una vez o muchas veces, pero también ninguna.
- `"a+"` → Usaremos el símbolo de suma para indicar que otro símbolo debe aparecer al menos una vez, pudiendo repetirse cuantas veces quiera.
- `"a{1,4}”` → Usando las llaves, podemos indicar el número mínimo y máximo de veces que un símbolo podrá repetirse. El primer número del ejemplo es el número 1, y quiere decir que la letra "a" debe aparecer al menos una vez. El segundo número, 4, indica que como máximo puede repetirse cuatro veces.
- `"a{2,}"` → También es posible indicar solo el número mínimo de veces que un carácter debe aparecer (sin determinar el máximo), haciendo uso de las llaves, indicando el primer número y poniendo la coma (no la olvides).
- `"a{5}"` → A diferencia de la forma anterior, si solo escribimos un número entre llaves, sin poner la coma detrás, significará que el símbolo debe aparecer un número exacto de veces. En este caso, la "a" debe aparecer exactamente 5 veces.
- `"[a-z]{1,4}[0-9]+"` → Los indicadores de repetición se pueden usar también con corchetes, dado que los corchetes representan, básicamente, un símbolo. En el ejemplo anterior se permitiría de una a cuatro letras minúsculas, seguidas de al menos un dígito numérico.

Para su uso, Java ofrece las clases Pattern y Matcher contenidas en el paquete `java.util.regex.*`. La clase Pattern se utiliza para procesar la expresión regular y "compilarla", lo cual significa verificar que es correcta y dejarla lista para su utilización. La clase Matcher sirve para comprobar si una cadena cualquiera sigue o no un patrón.

```
Pattern p=Pattern.compile("[01]+");
Matcher m=p.matcher("00001010");
if(m.matcher()) {
  System.out.println("Si, contiene el patrón");
} else {
  System.out.println("No, no contiene el patrón");
}
```

En el ejemplo, el método estático compile de la clase Pattern permite crear un patrón, dicho método compila la expresión regular pasada por parámetro y genera una instancia de Pattern (p en el ejemplo). El patrón p podrá ser usado múltiples veces para verificar si una cadena coincide o no con el patrón, dicha comprobación se hace invocando el método matcher, el cual combina el patrón con la cadena de entrada y genera una instancia de la clase Matcher (m en el ejemplo).

La clase Matcher contiene el resultado de la comprobación y ofrece varios métodos para analizar la forma en la que la cadena ha encajado con un patrón:

- `m.matches()` → Devolverá true si toda la cadena (de principio a fin) encaja con el patrón o false en caso contrario.
- `m.lookingAt()` → Devolverá true si el patrón se ha encontrado al principio de la cadena. A diferencia del método matches(), la cadena podrá contener al final caracteres adicionales a los indicados por el patrón, sin que ello suponga un problema.
- `m.find()` → Devolverá true si el patrón existe en algún lugar la cadena (no necesariamente toda la cadena debe coincidir con el patrón) y false en caso contrario, pudiendo tener más de una coincidencia. Para obtener la posición exacta donde se ha producido la coincidencia con el patrón podemos usar los métodos `m.start()` y `m.end()`, para saber la posición inicial y final donde se ha encontrado. Una segunda invocación del método `find()` irá a la segunda coincidencia (si existe), y así sucesivamente. Podemos reiniciar el método `find()`, para que vuelva a comenzar por la primera coincidencia, invocando el método `m.reset()`.

Veamos algunas construcciones adicionales que pueden ayudarnos a especificar expresiones regulares más complejas:

- `"[^abc]"` → El símbolo `"^"`, cuando se pone justo detrás del corchete de apertura, significa "negación". La expresión regular admitirá cualquier símbolo diferente a los puestos entre corchetes.
- `"^[01]+$"`.→ Cuando el símbolo `"^"` aparece al comienzo de la expresión regular, permite indicar comienzo de línea o de entrada. El símbolo `"$"` permite indicar fin de línea o fin de entrada. Usándolos podemos verificar que una línea completa (de principio a fin) encaje con la expresión regular, es muy útil cuando se trabaja en modo multilínea y con el método `find()`.

- `"."` → El punto simboliza cualquier carácter.
- `"\\d"` → Un dígito numérico.
- `"\\D"` → Cualquier cosa excepto un dígito numérico.
- `"\\s"` → Un espacio en blanco (incluye tabulaciones, saltos de línea y otras formas de espacio).
- `"\\S"` → Cualquier cosa excepto un espacio en blanco.
- `"\\w"` → Cualquier carácter que podrías encontrar, en una palabra.

Los paréntesis permiten indicar repeticiones para un conjunto de símbolos, por ejemplo: `"(#[01]){2,3}"` 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.

```
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("Numero:"+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. La excepción son las comillas, que se pondrían con una sola barra `"\""`.