6.C. Expresiones regulares.

1. Expresiones regulares.

1.2. Expresiones regulares (II).

¿Y cómo uso las expresiones regulares en un programa? Pues de una forma sencilla. 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. Veamoslo con un ejemplo:

```
Pattern p=Pattern.compile("[01]+");
```

```
Matcher m=p.matcher("00001010");
```

```
if (m.matches()) 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. En este caso, cualquier símbolo diferente de "a", "b" o "c".

- "`^[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. Equivale a "`[0-9]`".

- "`\\D`". Cualquier cosa excepto un dígito numérico. Equivale a "`[^0-9]`".

- "`\\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. Equivale a "`[a-zA-Z_0-9]`".

Autoevaluación

¿En cuáles de las siguientes opciones se cumple el patrón "`A.\\d+`"?

- [ ] "GA-99" si utilizamos el método find.
- [ ] "GAX99" si utilizamos el método lookingAt.
- [ ] "AX99-" si utilizamos el método matches.
- [ ] "A99" si utilizamos el método matches.

Resolver