Expresiones regulares

Una expresión regular (regex), es una secuencia de caracteres que forma un patrón de búsqueda.

Una expresión regular, nos permitirá **buscar patrones** en una cadena de texto para, por ejemplo, encontrar cuántas veces se repite una palabra en un texto. También nos permitirán comprobar que una cadena tiene una **determinada estructura**, por ejemplo, que el nombre de archivo que nos proponen tiene una determinada extensión, comprobar que un email está bien escrito, que el formato del DNI es correcto, etc...

Las expresiones regulares pueden contener, **además de letras y números**, los siguientes caracteres:

```
1    $, ^, ., *, +, ?, [, ], {, }, | , ( , )
```

- **[abc]**

  Uno de los tres caracteres

- **[**$\wedge abc$]

  Cualquier carácter excepto a,b,c

- **[a-z]**

  Carácter de a a z

- **[a-z0-9]**

  Carácter de a a z, y de 0 a 9.

- **.**

  Cualquier carácter

- $\backslash d$

  Carácter numérico. Equivale a [0-9]

- $\backslash D$

  Carácter no numérico

- $\backslash s$

  Carácter blanco, tabulador, salto de línea, etc.

- $\backslash S$

  Carácter no blanco, tabulador, salto de línea, etc.

- \w
  Carácter alfanumérico, o símbolo de subrayado.
- \W
  Carácter no alfanumérico, ni símbolo de subrayado
- ^
  Inicio de línea
- **$**
  Fin de línea
- **XY**
  X seguido de Y
- **X** | Y
  X o Y
- **X?**
  X una vez, o ninguna.
- **X***
  X cero o más veces.
- **X+**
  X una o más veces.
- **X{n}**
  X repetido n veces.
- **X{n,}**
  X repetido n veces o más.
- **X{n,m}**
  X repetido de n a m veces.
- **( )**
  Los paréntesis se usan para establecer un grupo de caracteres.

El paquete **java.util.regex** está formado por dos clases, la clase **Matcher** y la clase **Pattern** y por una excepción, **PatternSyntaxException**.

La clase **Pattern** (patrón) es la representación compilada de una expresión regular, o lo que es lo mismo, representa a la expresión regular, que en el paquete java.util.regex necesita estar compilada.

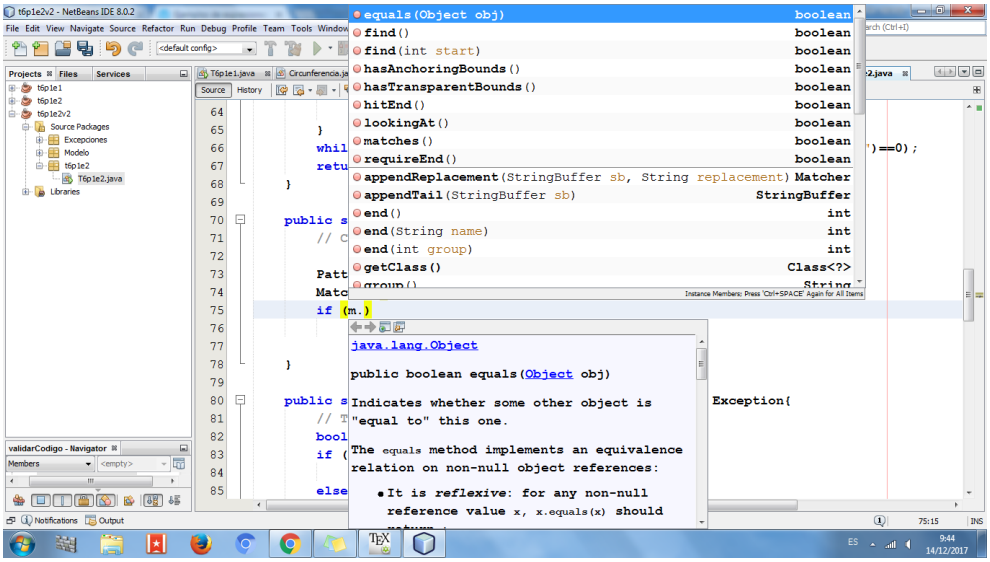
Para crear un patrón necesitamos compilar una expresión regular mediante el método compile.

```
1  Pattern patron = Pattern.compile("^[0-9]{5}$");
```

Una vez definido el patrón con Pattern, tenemos que crearnos un objeto Matcher.

```
1  String cadena;
2  Matcher encaja = patron.matcher(cadena);
```

Creado el objeto Matcher son muchas las operaciones que podemos realizar. Posiblemente el método **matches** sea el más utilizado. Este método intenta encajar toda la secuencia en el patrón.



Ejemplos:

```
1  Comprobar si el String cadena contiene exactamente el patrón (matches) abc
2
3       Pattern pat = Pattern.compile("^abc$");
4       Matcher mat = pat.matcher(cadena);
5       if (mat.matches()) {
6           System.out.println("SI");
```

```
7       } else {
8           System.out.println("NO");
9       }
```

```
Comprobar si el String cadena contiene abc

      Pattern pat = Pattern.compile(".*abc.*");
      Matcher mat = pat.matcher(cadena);
      if (mat.matches()) {
          System.out.println("SI");
      } else {
          System.out.println("NO");
      }
```

```
Comprobar si el String cadena empieza por abc

      Pattern pat = Pattern.compile("^abc.*");
      Matcher mat = pat.matcher(cadena);
      if (mat.matches()) {
          System.out.println("Válido");
      } else {
          System.out.println("No Válido");
      }
```

```
Comprobar si el String cadena empieza por abc ó Abc

      Pattern pat = Pattern.compile("^[aA]bc.*");
      Matcher mat = pat.matcher(cadena);
      if (mat.matches()) {
          System.out.println("SI");
      } else {
          System.out.println("NO");
      }
```

```
Comprobar si el String cadena está formado por un mínimo de 5 letras
    mayúsculas o minúsculas y un máximo de 10.

      Pattern pat = Pattern.compile("[a-zA-Z]{5,10}");
      Matcher mat = pat.matcher(cadena);
      if (mat.matches()) {
          System.out.println("SI");
      } else {
          System.out.println("NO");
      }
```

```
public static void validarCodigo(String codigo)throws Exception{

        // Cinco dígitos numéricos

        Pattern patron = Pattern.compile("^[0-9]{5}$");
        Matcher m = patron.matcher(codigo);
        if (!m.matches())
            throw new DatoNoValido(1);
    }
```

Todo lo anterior está orientado a la búsqueda de patrones en cadenas de caracteres, pero puede que queramos llegar mas allá, que lo que queramos sea reemplazar una cadena de caracteres que se corresponda con un patrón por otra cadena. Un método que consigue esto es **replaceAll**.

El ejemplo sustituye todas las apariciones que concuerden con el patrón a*b por la cadena -.

```
1  // se importa el paquete java.util.regex
2  import java.util.regex.*;
3
4  public class EjemploReplaceAll{
5
6    public static void main(String args[]){
7
8    // compilamos el patron
9    Pattern patron = Pattern.compile("a*b");
10
11   // creamos el Matcher a partir del patrón, la cadena como parámetro
12   Matcher encaja = patron.matcher("aabmanoloaabmanoloabmanolob");
13
14   // invocamos el método replaceAll
15   String resultado = encaja.replaceAll("-");
16   System.out.println(resultado); // -manolo-manolo-manolo-
17   }
18 }
```

El siguiente ejemplo trata de validar una cadena que supuestamente contiene una dirección de correo, lo hace con cuatro comprobaciones, con un patrón cada una, la primera que no contenga como primer carácter una @ o un punto, la segunda que no comience por www. , que contenga una y solo una @ y la cuarta que no contenga caracteres ilegales.

```
1  import java.util.regex.*;
2
3  public class ValidacionEmail {
4     public static void main(String[] args) throws Exception {
5        String input = "www.?regular.com";
6
7        // comprueba que no empieze por punto o @
8        Pattern p = Pattern.compile("^.|^@");
9        Matcher m = p.matcher(input);
10       if (m.find())
11          System.err.println("Las direcciones email no empiezan por punto
                 o @");
12
13       // comprueba que no empieze por www.
14       p = Pattern.compile("^www.");
15       m = p.matcher(input);
16       if (m.find())
17         System.out.println("Los emails no empiezan por www");
18
19       // comprueba que contenga @
20       p = Pattern.compile("@");
21       m = p.matcher(input);
```

```
22        if (!m.find())
23          System.out.println("La cadena no tiene arroba");
24
25        // comprueba que no contenga caracteres prohibidos
26        p = Pattern.compile("[^A-Za-z0-9.@_-~#]+");
27        m = p.matcher(input);
28        StringBuffer sb = new StringBuffer();
29        boolean resultado = m.find();
30        boolean caracteresIlegales = false;
31
32        while(resultado) {
33           caracteresIlegales = true;
34           m.appendReplacement(sb, "");
35           resultado = m.find();
36        }
37
38        // Añade el ultimo segmento de la entrada a la cadena
39        m.appendTail(sb);
40
41        input = sb.toString();
42
43        if (caracteresIlegales) {
44           System.out.println("La cadena contiene caracteres ilegales");
45        }
46     }
47  }
```

Otras expresiones regulares que nos permiten validar un email.

```
1  String regex = "[A-Za-z]+@[a-z]+\\.[a-z]+";
2
3  String emailPattern =
       "^[_a-z0-9-]+(\\.[_a-z0-9-]+)*@[a-z0-9-]+(\\.[a-z0-9-]+)*(\\.[a-z]{2,4})$";
```

- ^ especifica el inicio de la entrada.
- **([_a-z0-9-])+** primer grupo. Se refiere a la aparición de uno o más caracteres compuestos por guión bajo, letras, números y guiones.
- **(**
  .[_a-z0-9-])* segundo grupo. Puede ser opcional y repetible, se refiere a la aparición de un punto seguido de uno o más caracteres compuestos por guión bajo, letras, números y guiones.
- **@** carácter arroba.
- **([a-z0-9-])+** tercer grupo. Especifica la aparición de uno o más caracteres compuestos por letras, números y guiones.
- **(**
  .[a-z0-9-])* cuarto grupo. Especifica un punto seguido de uno o más caracteres compuestos por letras, números y guiones.

- **(**
  .[a-z]2,4) quinto grupo. Especifica un punto seguido de entre 2 y 4 letras, con el fin de considerar dominios terminados, por ejemplo, en .co y .info.
- **$** especifica el fin de la entrada.

Expresiones regulares relacionadas con fechas.

```
1  String regexp = "\\d{1,2}/\\d{1,2}/\\d{4}";
2
3  // Lo siguiente devuelve true
4  System.out.println(Pattern.matches(regexp, "11/12/2018"));
5  System.out.println(Pattern.matches(regexp, "1/12/2018"));
6  System.out.println(Pattern.matches(regexp, "11/2/2018"));
7
8
9  // Los siguientes devuelven false
10 System.out.println(Pattern.matches(regexp, "11/12/18"));  // El año no
       tiene cuatro cifras
11 System.out.println(Pattern.matches(regexp, "11//2018"));  // el mes no
       tiene una o dos cifras
12 System.out.println(Pattern.matches(regexp, "11/12/18perico"));  // Sobra
       "perico"
13
14
15
16 String literalMonthRegexp =
       "\\d{1,2}/(ene|feb|mar|abr|may|jun|jul|ago|sep|oct|nov|dic)/\\d{4}";
17
18 // Lo siguiente devuelve true
19 System.out.println(Pattern.matches(literalMonthRegexp, "11/dic/2018"));
20 System.out.println(Pattern.matches(literalMonthRegexp, "1/nov/2018"));
21 System.out.println(Pattern.matches(literalMonthRegexp, "1/AGO/2018"));
       // Mes en mayúsculas
22 System.out.println(Pattern.matches(literalMonthRegexp, "21/Oct/2018"));
       // Primera letra del mes en mayúsculas.
23
24 // Los siguientes devuelven false
25 System.out.println(Pattern.matches(literalMonthRegexp, "11/abc/2018"));
       // abc no es un mes
26 System.out.println(Pattern.matches(literalMonthRegexp, "11//2018"));
       // falta el mes
27 System.out.println(Pattern.matches(literalMonthRegexp,
       "11/jul/2018perico"));   // sobra perico
```