EXPRESIONES REGULARES

Sitio solvente **http://docs.oracle.com/javase/tutorial/essential/regex/intro.html**
pero muy extenso para nosotros ...

¿Qué es una expresión regular?

Sin rigor, podemos decir que una expresión regular(ER) es "una especie de fórmula" o "patrón" que describe un conjunto de Strings. La fórmula se basa en la utilización de caracteres que tienen un significado especial.

Ejemplo:
la expresión regular *-?\d+* representa a toda aquello que:

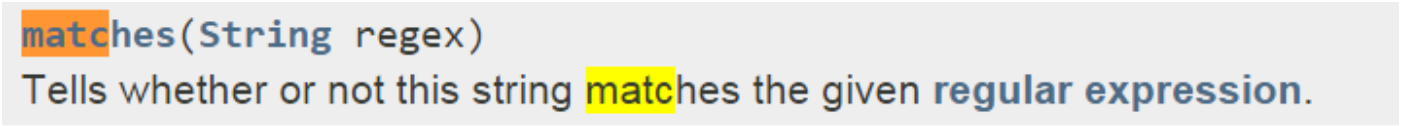
- -?: comienza con un carácter -, pero esto es opcional ya que al - le sigue un ?
- \d+: a continuación debe venir uno o varios dígitos :\d significa dígito y + significa 1 o más

Observa que los caracteres ?, \, d y + son caracteres con un significado especial en una expresión regular. En cambio el caracter - no tiene un significado especial. Hay toda una sintaxis para escribir una ER
El conjunto de Strings que son representados por la expresión anterior es:
-0,0,-1,1,-2,2,-888888,564, etc. o sea el conjunto de Strings que representan todos los números enteros negativos y positivos pero los positivos no pueden empezar por +.

Lo anterior es sólo un ejemplo, para hacerse una idea intuitiva de lo que es una ER. Iremos viendo la sintaxis de una expresión regular progresivamente. Por el momento vamos a probar cómo usar la expresión anterior.

EL MÉTODO matches() de la clase STRING

La forma más simple de utilizar expresiones regulares en java es a través del método matches() de la clase String.
Si consultas el api



Al método matches has de pasarle un String que describe la expresión regular. Como el carácter \, es un carácter con un significado especial en las expresiones regulares, pero también lo es en los string(caracter de escape), debemos anteceder la \ con otra \ y por ello el string que pasamos es

"-?\\d+"
Recuerda que la \ en un string se "escapaba" precediéndola de otra barra slash.

```
public class App {
    public static void main(String[] args) {
        System.out.println("-1234".matches("-?\\d+"));
        System.out.println("1234".matches("-?\\d+"));
        System.out.println("+1234".matches("-?\\d+"));
    }
}
```

```
run:
true
true
false
```

Sobre la traducción de match
Tiene muchos significados. En esta unidad es apropiado traducirla por "emparejar" o "encajar". Diremos pues que una expresión regular "empareja" o "encaja" *contra* un String.

ALGO DE SINTAXIS

Vemos aspectos muy habituales pero teniendo en cuenta que faltan muchos detalles.

- Concatenacion. Simplemente por secuencia de caracteres
  Ejemplo: ab el string ab

Sintaxis más básica de las expresiones regulares.

- Los corchetes [] indican opcionalidad
  Ejemplo: [abc][12] representa a todos los strings de dos caracteres en el que el primero es a, b o c y el segundo 1 o 2 o sea representa el conjunto de Strings: a1,a2,b1,b2,c1 y c2
- Otra forma de opcionalidad
  Ejemplo: a|b el string de un carácter que puede ser a o b
- Los guiones pueden representar rangos, aunque en otros contextos es un carácter normal como vimos en el ejemplo de la introducción `-?\\d+`
  Ejemplo:[a-z]1 todos los strings con 2 caracteres el primero es una letra de la a a la z y el segundo un 1
- . (el carácter punto) indica cualquier símbolo "encaja" con cualquier carácter
- Los paréntesis () para hacer grupos de caracteres
- El ^ tiene varios significados según donde se use. Uno habitual es negar cuando se usa dentro de corchetes. Sin corchetes tiene otro significado
  Ejemplo:[^abc]x todos los strings de 2 caracteres que NO empiezan por a, b o c y a continuación viene una x, es decir: tx,rx,wx pero no valdrían ax, bx, cx,...
- Hay algún símbolo más.

Ejemplo en Java de los símbolos anteriores. Todo da true en el ejemplo

```
class App{
    public static void main(String[] args) {
        System.out.println("ho".matches("(h|z)o"));
        System.out.println("hola3".matches("hola[0-9]"));
        System.out.println("c2".matches("[a-z][1-6]"));
        System.out.println("@".matches("."));
        System.out.println("@".matches(".."));//false
        System.out.println("@@".matches(".."));
    }
}
```

Ejercicio: cambia sobre el ejemplo anterior los strings y observa si obtienes el valor esperado

Símbolos meta caracteres: representan a todos los caracteres de un tipo

\d *Dígito*. Equivale a [0-9]

\D lo contrario de \d o sea no dígito [^0-9]

\s *Espacio en blanco*. Equivale a [\t\n\x0b\r\f]. En programación a veces espacio en blanco *white space* se refiere no sólo al espacio en blanco propiamente dicho si no a un conjunto de caracteres que separan con uno o varios blancos

\S lo contrario de \s

\w Una letra mayúscula o minúscula, un dígito o el carácter _ Equivale a [a-zA-Z0-9_]

\W lo contrario de \w

ojo :"\" , es un carácter especial en expresiones regulares pero también es un carácter de escape en Strings Java

Como la ER la indicamos muchas veces dentro de un String Java hay conflicto ya que \ tiene un significado especial en String java y en ER. La solución es escapara la \ escribiendo \\ dentro del string java

```
class App{
```

```
   public static void main(String[] args) {
      System.out.println("h1".matches("\\w\\d"));
      System.out.println("hola3".matches("hola\\d"));

   }
}
```

Símbolos cuantificadores

Sirven para indicar repetición

- Los corchetes {}
  - {X} lo que va justo antes de las llaves se repite X veces
    Ejemplo: a{2}bc encaja con aabc
  - {X,Y} lo que va justo antes de las llaves se repite mínimo X veces y máximo Y veces.
    Ejemplo: a{1,3}bc encaja con abc, aabc, aaabc
  - También podemos poner {X,} indicando que se repite un mínimo de X veces sin límite máximo.
- * Indica 0 ó más veces. Equivale a {0,}
  ejemplo: "a*bc" encaja con *bc, abc, aabc, aaabc, ...*
- + Indica 1 ó más veces. Equivale a {1,}
  ejemplo: "a+bc" encaja con *abc, aabc, aaabc, …¡pero no con bc!*
- ? Indica 0 ó 1 veces. Equivale a {0,1}
  ejemplo: "a?bc" encaja con *abc* y con *bc*

  Además el ? tiene un uso adicional cuando acompaña a un cuantificador como + para controlar su comportamiento expansivo como veremos más adelante

```
class App{
   public static void main(String[] args) {
      System.out.println("aaaaahola".matches("a+hola"));
      System.out.println("hola".matches("a+hola"));//false
      System.out.println("hola".matches("a*hola"));
      System.out.println("hola".matches("a?hola"));
      System.out.println("ahola".matches("a?hola"));
   }
}
```

Otros ejemplos:
#[01]{2,3} encaja con #00 #000 #10 etc.
Si añado paréntesis ...
(#[01]){2,3} encaja con #0#0 #0#0#0 #1#0 etc.(observa como se repite también el #)

[a-z]{1,4}[0-9]+ encaja con todo String que comience por 1, 2, 3 o 4 letras minúsculas seguidas de al menos un dígito numérico: zd1, a3333339333353333333,abcd99,etc

Relacionando String, arrays y expresiones regulares. El método split()

Las expresiones regulares no se utilizan únicamente con el método matches(). Su uso

principal es con la clase Pattern que veremos en el próximo boletín pero además también se utiliza en métodos que realmente ya utilizaste:

- useDelimiter() de la clase Scanner.(ver API). Lo usamos indicando un caracter pero
- split() de la clase String y Pattern

Ejemplo:

```
class App {

   public static void main(String[] args) {
      String texto="a:b:c:d;e:f,g:h";
      String []partes=texto.split(":");
      System.out.print("Caso1: |");
      for(String s:partes)
       System.out.print(s+ "|");
      System.out.println();

      texto="1,2:3,4;5,6";
      partes=texto.split(";");
      System.out.print("Caso2: |");
      for(String s:partes)
       System.out.print(s+ "|");
      System.out.println();

      texto="hola7adios22chao3315que tal 9pues vale";
      partes=texto.split("\\d+");
      System.out.print("Caso3: |");
      for(String s:partes)
       System.out.print(s+ "|");
      System.out.println();


      //ojo con algunos caracteres que en expresiones regulares tiene significado especial como .
      //con . como expresion todo la cadena encaja y no devuelve nada por que no hay trozos
      texto="hola.adios,2:3,4;5,6";
      partes=texto.split(".");
      System.out.print("Caso4: |");
      for(String s:partes)
       System.out.print(s+ "|");
      if(partes.length==0)
         System.out.println("no hay partes, tomo el . como comodin");

      partes=texto.split("\\.");
      System.out.print("Caso5: |");
      for(String s:partes)
       System.out.print(s+ "|");
   }
}
```

Ejercicio U6_B1A_E1: escribe una expresión regular que reconozca si una cadena de texto se corresponde con un (único) número binarios y pruébala en un main

Ejercicio U6_B1A_E2: valida strings que contienen DNIs usando una expresión regular que de por bueno el siguiente formato: de 1 a 8 dígitos seguidos por una letra mayúscula o minúscula.

Prueba la ER con

```
public class App {
   public static void main(String[] args) {
      String er=ESCRIBE AQUÍ TU EXPRESIÓN REGULAR
      String[] dnis = {"1a","a1","12345678A", "123456789A", "123456789", "0000001R", "00000001R", "", "123abv11a"};
      for (String dni : dnis) {
```

```
            System.out.println(dni+ " es "+ dni.matches(er));
        }
    }
}
```

Ejercicio U6_B1A_E3: El argumento args[0] es una cadena que contiene dnis separados por un espacio en blanco. Se analiza cada DNI para ver si es correcto con la ER del ejercicio anterior. Utiliza también el método split().

Ejercicio U6_B1A_E4: Crea un validador de direcciones IP. Recuerda que las direcciones IP se especifican en decimal como 4 grupos de números separados por ".” Cada grupo puede contener un número decimal de 0 a 255. Por ejemplo son IP válidas:

- 0.1.2.3
- 255.255.255.255
- 9.234.1.199

Son inválidas:

- 0.1.2.
- 0.1.2.3.
- 256.1.2.3
- 192.168.1.01 (no queremos ceros a la izquierda)

Dentro de las IPs válidas estas pueden tener varias funciones: direcciones de host, direcciones de red, direcciones broadcast, máscaras, pero esto no influye en este ejercicio. Ejemplo de ejecución: