Tema 7.- Expresiones Regulares

Contenido

1. Expresiones Regulares en Java

http://download.oracle.com/javase/tutorial/essential/regex/

Una expresión regular es una forma abreviada de realizar operaciones o búsquedas en la que se define un patrón para cadenas de caracteres y así realizar dichas operaciones o búsquedas con el comportamiento variable de los símbolos establecidos en la sintaxis de las expresiones regulares.

En Java, las expresiones regulares son algo que se usa desde la versión 1.4 del JDK de Sun y se incluye el paquete `java.util.regex`, que proporciona una serie de clases para poder hacer uso de la potencia de este tipo de expresiones en Java.

Una expresión regular define un patrón de búsqueda para cadenas de caracteres.

La podemos utilizar para comprobar si una cadena contiene o coincide con el patrón. El contenido de la cadena de caracteres puede coincidir con el patrón 0, 1 o más veces.

El patrón se busca en el String de izquierda a derecha. Cuando se determina que un carácter cumple con el patrón este carácter ya no vuelve a intervenir en la comprobación.Es decir, con la expresión regular "010" dentro del String "010101010" la encontraremos solo dos veces: "010101010".

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` recogerá en cada objeto la representación compilada de una expresión regular. Es decir, una expresión regular debe estar representada por su expresión compilada en el paquete `java.util.regex` para ser utilizada. `Pattern`, en castellano significa patrón.
- Con la clase `Matcher` se crea un objeto a partir de un patrón mediante la invocación del método `Pattern.matcher`. Este objeto es el que nos permite realizar operaciones sobre la secuencia de caracteres que queremos validar o en la secuencia de caracteres en la que queremos buscar. En castellano lo más parecido a esto es la palabra encajador.

Por lo tanto tenemos patrones que deben ser compilados, a partir de estos creamos objetos de la clase `Matcher` (encajadores) para poder realizar las operaciones sobre la cadena en cuestión.

1.1. Clase Pattern

Un objeto de esta clase representa la expresión regular. Contiene el método `compile(String regex)` que recibe como parámetro la expresión regular y devuelve un objeto de la clase `Pattern`, que será el patrón para la verificación.

```
Pattern patron = Pattern.compile("camion");
```

El método **static** `compile` devuelve la expresión regular, que hemos pasado como parámetro, compilada y la guarda en el objeto `patron` de tipo `Pattern` que hemos definido.

El método `matcher` crea un objeto de tipo `Matcher` a partir del objeto `patron` y la cadena que se quiere analizar. El método `split` divide una cadena dada en partes que cumplan el patrón compilado.

```
Matcher encaja = patron.matcher(cadena);
```

1.2. Clase Matcher

Esta clase compara el String y la expresión regular. Contiene el método `matches(CharSequence input)` que recibe como parámetro el String a validar y devuelve true si coincide con el patrón.

Un objeto `Matcher` se genera a partir de un objeto `Pattern` por medio del método `matcher`:

```
Pattern patron = Pattern.compile("camion");
Matcher encaja = patron.matcher(cadena);
```

Una vez que tenemos el objeto `encaja` creado, podemos realizar tres tipos de operaciones sobre una cadena de caracteres.

- Una es a través del método **`matches()`** que intenta encajar **TODA LA SECUENCIA** en el patrón (para el patrón "camion" la cadena "camion" encajaría, la cadena "mi camion es verde" no encajaría). Devuelve *true* si toda la cadena (*de principio a fin*) encaja con el patrón o *false* en caso contrario.
- Otra es a través del método **`lookingAt()`**, intenta encajar el patrón en la cadena (para el patrón "camion" tanto la cadena "camion" como la cadena "mi camion es verde" encajaría). Devuelve true si el patrón se ha encontrado **AL PRINCIPIO** de la cadena, false en caso contrario.
- Otra es la proporcionada por el método **`find()`,**que devuelve **`true`** si el patrón existe en algún lugar la cadena y **`false`** en caso contrario. Para obtener la posición exacta donde se ha producido la coincidencia con el patrón podemos usar:
  - el método **`start`**(), que devuelve el índice del **primer** carácter de la ocurrencia en la secuencia.
  - el método **`end()`**, que devuelve el índice del carácter que sigue al **último** de la ocurrencia.

  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 **`reset`**`()`.

Todo lo anterior está orientado a la búsqueda de patrones en cadenas de caracteres, pero puede que queramos llegar más allá, que lo que queramos sea reemplazar una cadena de caracteres que se corresponda con un patrón por otra cadena. Por ejemplo, un método que consigue esto es `replaceAll` de la clase `String` que reemplaza toda ocurrencia del patrón en la cadena por la cadena que se le suministra.

Ejemplos

- El siguiente es un ejemplo del uso del método replaceAll sobre una cadena. El ejemplo sustituye todas las apariciones que concuerden con el patrón "a*b" por la cadena "-".

```
// se importa el paquete java.util.regex
import java.util.regex.*;

public class EjemploReplaceAll{
   public static void main(String args[]){
      // compilamos el patron
      Pattern patron = Pattern.compile("a*b");
      // creamos el Matcher a partir del patron, la cadena como parámetro
      Matcher encaja = patron.matcher("aabmanoloaabmanoloabmanolob");
      // invocamos el método replaceAll
      String resultado = encaja.replaceAll("-");
      System.out.println(resultado); // muestra: -manolo-manolo-manolo-
```

```
    }
}
```

- El siguiente ejemplo trata de validar una cadena que supuestamente contiene un email. 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:

```
import java.util.regex.*;

public class ValidacionEmail {
   public static void main(String[] args) throws Exception {
      String input = "www.?regular.com";
      // comprueba que no empiece por punto o @
      Pattern p = Pattern.compile("^[.|@]");
      Matcher m = p.matcher(input);
      if (m.find()){
         System.err.println("Las direcciones email no empiezan por punto o @");
      }
      // comprueba que no empiece por www.
      p = Pattern.compile("^www.");
      m = p.matcher(input);
      if (m.find()){
        System.out.println("Los emails no empiezan por www");
      }
      // comprueba que contenga @
      p = Pattern.compile("@");
      m = p.matcher(input);
      if (!m.find()){
       System.out.println("La cadena no tiene arroba");
      }
      // comprueba que no contenga caracteres prohibidos
      p = Pattern.compile("[^A-Za-z0-9.@_-~#]+");
      m = p.matcher(input);
      StringBuffer sb = new StringBuffer();
      boolean resultado = m.find();
      boolean caracteresIlegales = false;

      while(resultado) {
         caracteresIlegales = true;
         m.appendReplacement(sb, "");
         resultado = m.find();
      }

      // Añade el último segmento de la entrada a la cadena
      m.appendTail(sb);

      input = sb.toString();

      if (caracteresIlegales) {
         System.out.println("La cadena contiene caracteres ilegales");
      }
   }
}
```

La salida es:

```
Los emails no empiezan por www
La cadena no tiene arroba
La cadena contiene caracteres ilegales
```

- El siguiente ejemplo permite validar diferentes expresiones regulares

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ExpresionRegular {
   public static void main(String[] args) {

      try {
         BufferedReader teclado;
         teclado = new BufferedReader (new InputStreamReader(System.in));

         String texto =
      "Cuando utilice cart@uchos.es de impresión que no de email@mail.tk, "+
      "es que algunas características impresión de los datos del ej. http://blogspot.com  " +
      "volumen restante, estén disponibles por 192.168.1.1 usar suministro que no sea de api.";

         System.out.println("Inserta expresion regular: ");
         String ex_regular = teclado.readLine();

         Pattern pat=Pattern.compile(ex_regular);
         Matcher mat=pat.matcher(texto);

         while(mat.find()){
            String aux = mat.group();
            System.out.println("Encontrado: " + aux);
         }

      }
      catch (IOException ex) {System.out.println("Error");}
   }
}
```

Resultado 1:
Inserta expresion regular: [0-9]+ (aquí ponemos manualmente la Expresión Regular deseada)
Encontrado: 192
Encontrado: 168
Encontrado: 1
Encontrado: 1

Resultado 2:
Inserta expresion regular: \b(?:\d{1,3}\.){3}\d{1,3}\b
Encontrado: 192.168.1.1

Resultado 3:
Inserta expresion regular: ((\w)+(\.\w+)?)+@\w+\.\w+
Encontrado: cart@uchos.es
Encontrado: email@mail.tk

La expresión regular que permite verificar si una cadena contiene un DNI o un NIE es la siguiente: `"[XYxy]?[0-9]{1,9}[A-Za-z]"`; aunque no es la única solución.

2. Descripción de las expresiones regulares

Símbolos comunes en expresiones regulares

Expresión	Descripción
.	Un punto indica cualquier carácter, un solo carácter.
^expresión	El símbolo ^ indica el principio del String. En este caso el String debe contener la expresión al principio. "^\d$" un único dígito
expresión$	El símbolo $ indica el final del String. En este caso el String debe contener la expresión al final. "\.$" lugares donde . finaliza línea
[abc]	Los corchetes representan una definición de conjunto sobre una posición. En este ejemplo el String debe contener una de las letras a ó b ó c.
[abc][12]	El String debe contener una letra de entre a ó b ó c seguidas de un 1 ó un 2
[^abc]	El símbolo ^ dentro de los corchetes indica negación. En este caso el String debe contener cualquier carácter excepto a ó b ó c.
[a-z1-9]	Rango. Indica las letras minúsculas desde la a hasta la z (ambas incluidas) y los dígitos desde el 1 hasta el 9 (ambos incluidos)
A|B	El carácter | es un OR. A ó B
AB	Concatenación. A seguida de B
()	Agrupa caracteres. Los caracteres especiales conservan su valor. | bloques opcionales

Meta caracteres

Expresión	Descripción
\d	Dígito. Equivale a [0-9]
\D	No dígito. Equivale a [^0-9]
\s	Espacio en blanco. Equivale a [\t\n\x0b\r\f]
\S	No espacio en blanco. Equivale a [^\s]
\w	Una letra mayúscula o minúscula, un dígito o el carácter _ Equivale a [a-zA-Z0-9_]
\W	Equivale a [^\w]
\b	Límite de una palabra. (delimitada por espacios en blanco, puntuación o inicio/final de cadena)

En Java debemos usar una doble barra invertida \\ Por ejemplo para utilizar \w tendremos que escribir \\w. Si queremos indicar que la barra invertida es un carácter de la expresión regular tendremos que escribir \\ . El punto como carácter será \.

Cuantificadores

Expresión	Descripción
{X}	Indica que lo que va justo antes de las llaves se repite X veces
{X,Y}	Indica que lo que va justo antes de las llaves se repite mínimo X veces y máximo Y veces. También podemos poner {X,} indicando que se repite un mínimo de X veces sin límite máximo.
*	Indica 0 ó más veces. Equivale a {0,}
+	Indica 1 ó más veces. Equivale a {1,}
?	Indica 0 ó 1 veces. Equivale a {0,1} . Se puede aplicar a bloques que estén entre paréntesis

3. Variables String y métodos con expresiones regulares.

3.1. String.matches(regex)

Podemos comprobar si una cadena de caracteres cumple con un patrón usando el método `matches` de la clase String. Este método recibe como parámetro la expresión regular. Por ejemplo, para comprobar si el String *cadena* contiene un 1 y ese 1 no está seguido por un 2

```
    if (cadena.matches(".*1(?!2).*")) {
        System.out.println("SI");
    } else {
        System.out.println("NO");
    }
```

3.2. String.split(regex)

El método `split` de la clase String es la alternativa a usar StringTokenizer para separa cadenas. Este método divide el String en cadenas según la expresión regular que recibe. La expresión regular no forma parte del array resultante.

Ejemplo 1:

```
    String str = "blanco-rojo:amarillo.verde_azul";
    String [] cadenas = str.split("[-:._]");
    for(int i = 0; i<cadenas.length; i++){
        System.out.println(cadenas[i]);
    }
```

Muestra por pantalla:

```
blanco
rojo
amarillo
verde
azul
```

Ejemplo 2:

```
    String str = "esto es un ejemplo de como funciona split";
    String [] cadenas = str.split("(e[s|m])|(pl)");
    for(int i = 0; i<cadenas.length; i++){
        System.out.println(cadenas[i]);
    }
```

Salida:

```
to
 un ej
o de como funciona s
it
```

Herramientas online para probar expresiones regulares.

- **Txt2re**
- **Regexpal**
- **RegExr**
- **Regexper**