3.2 Expresiones regulares

1. Expresiones regulares

Una **expresión regular** es una secuencia de caracteres que forma un patrón de búsqueda proporcionando una manera muy flexible de buscar o reemplazar cadenas de texto.

En la clase String hay un método llamado ***matches*** que indica si la cadena coincide o no con la expresión regular que se le pasa por parámetro.

He aquí algunas de las construcciones de expresiones regulares que se encuentran en la API de Java en la clase *Pattern*:

- **Characters**

Construct	Matches
x	The character *x*
\\	The backslash character
\u*hhhh*	The character with hexadecimal value 0x*hhhh*
\t	The tab character ('\u0009')
\n	The newline (line feed) character ('\u000A')

```
package tema3_2_ExpresionesRegulares;

public class Characters {

    public void show() {

        System.out.println("a".matches("a"));//true
        System.out.println("b".matches("a"));//false
        System.out.println("A".matches("\u0041"));//true
        System.out.println("\n".matches("\n"));//true
        System.out.println("\t".matches("\t"));//true
        System.out.println("\\".matches("\\\\"));//true

    }
```

```
    public static void main(String[] args) {

        new Characters().show();

    }

}
```

Tal y como vimos en el *Tema 1.1 Introducción al lenguaje Java 10 Tipos de datos primitivos 10.4 Caracteres*, secuencias de escape es el conjunto de caracteres que en el código es interpretado con algún fin. En Java, la barra invertida `\` se denomina **carácter de escape**, el cual indica que el carácter puesto a continuación será convertido en carácter especial o, si ya es especial, dejará de ser especial. Por ejemplo, el carácter `n` no es especial pero con la `\` delante se convierte en especial ya que `\n` se interpreta como un salto de línea. La `\` es un carácter especial pero con otra `\` delante deja de ser especial y simplemente es una barra invertida.

En las expresiones regulares también se utiliza la barra invertida `\` como carácter de escape.

Veamos la última línea del código anterior `"\\".matches("\\\\"))` : la cadena `"\\"` es una barra invertida, y el argumento del matches `"\\\\"` son dos barras invertidas ya que la expresión regular de la barra invertida son dos barras tal y como podemos observar en la tabla anterior.

- **Logical operators**

Construct	Matches
XY	X followed by Y
X|Y	Either X or Y

```
package tema3_2_ExpresionesRegulares;

public class LogicalOperators {

    public void show() {

        System.out.println("hola".matches("hola"));//true
        System.out.println("hol".matches("hola"));//false

        System.out.println("hola".matches("hola|adios"));//true
        System.out.println("adios".matches("hola|adios"));//true
        System.out.println("hol".matches("hola|adios"));//false
        System.out.println("Adios".matches("hola|adios"));//false

    }

    public static void main(String[] args) {

        new LogicalOperators().show();

    }

}
```

- **Character classes**

Construct	Matches
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]]	a through z, and not m through p: [a-lq-z] (subtraction)

```
package tema3_2_ExpresionesRegulares;

public class CharactersClasses {

    public void show() {

        System.out.println("a".matches("[abc]"));//true
        System.out.println("d".matches("[abc]"));//false
        System.out.println("a".matches("[abc][abc]"));//false
        System.out.println("ac".matches("[abc][abc]"));//true
        System.out.println("ad".matches("[abc][abc]"));//false
        System.out.println(" a".matches("[abc ][abc]"));//true

        System.out.println("d".matches("[^abc]"));//true
        System.out.println("a".matches("[^abc]"));//false
        System.out.println("d".matches("[^abc][^abc]"));//false
        System.out.println("de".matches("[^abc][^abc]"));//true
        System.out.println("da".matches("[^abc][^abc]"));//false

        System.out.println("A".matches("[a-zA-Z]"));//true
        System.out.println("9".matches("[a-zA-Z]"));//false
        System.out.println("A".matches("[a-zA-Z][a-zA-Z]"));//false
        System.out.println("Az".matches("[a-zA-Z][a-zA-Z]"));//true
        System.out.println("A9".matches("[a-zA-Z][a-zA-Z]"));//false

        System.out.println("b".matches("[a-d[m-p]]"));//true
        System.out.println("n".matches("[a-d[m-p]]"));//true
        System.out.println("s".matches("[a-d[m-p]]"));//false

        System.out.println("d".matches("[a-z&&[def]]"));//true
        System.out.println("a".matches("[a-z&&[def]]"));//false

        System.out.println("d".matches("[a-z&&[^bc]]"));//true
        System.out.println("b".matches("[a-z&&[^bc]]"));//false

        System.out.println("d".matches("[a-z&&[^m-p]]"));//true
        System.out.println("n".matches("[a-z&&[^m-p]]"));//false
```

```
    }

    public static void main(String[] args) {

        new CharactersClasses().show();

    }

}
```

- **Predefined character classes**

Construct	Matches
.	Any character (may or may not match line terminators)
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A whitespace character: [\t\n\x0B\f\r]
\S	A non-whitespace character: [^\s]
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

Estas expresiones regulares van contenidas en una cadena. Entonces, aquellas expresiones regulares que lleven una barra invertida, como por ejemplo `\d`, tienen que llevar otra barra invertida `\` delante ya que en las cadenas una barra invertida se expresa como `\\`.

```
package tema3_2_ExpresionesRegulares;

public class PredefinedCharacterClasses {

    public void show() {

        System.out.println("a".matches("."));//true
        System.out.println("?".matches("."));//true
        System.out.println("\n".matches("."));//false

        System.out.println("2".matches("\\d"));//true
        System.out.println("a".matches("\\d"));//false

        System.out.println("2".matches("\\D"));//false
        System.out.println("a".matches("\\D"));//true
        System.out.println("a2".matches("\\D\\d"));//true

        System.out.println(" ".matches("\\s"));//true
        System.out.println("\t".matches("\\s"));//true
        System.out.println("\n".matches("\\s"));//true
        System.out.println("a".matches("\\s"));//false

        System.out.println(" ".matches("\\S"));//false
```

```
        System.out.println("a".matches("\\S"));//true
        System.out.println(" a".matches("\\s\\S"));//true

        System.out.println("_".matches("\\w"));//true
        System.out.println("B".matches("\\w"));//true
        System.out.println("9".matches("\\w"));//true
        System.out.println("?".matches("\\w"));//false

        System.out.println("a".matches("\\W"));//false
        System.out.println("*".matches("\\W"));//true

    }

    public static void main(String[] args) {

        new PredefinedCharacterClasses().show();

    }

}
```

- **POSIX character classes (US-ASCII only)**

Construct	Matches
\p{Lower}	A lower-case alphabetic character: [a-z]
\p{Upper}	An upper-case alphabetic character:[A-Z]
\p{ASCII}	All ASCII:[\x00-\x7F]
\p{Alpha}	An alphabetic character:[\p{Lower}\p{Upper}]
\p{Digit}	A decimal digit: [0-9]
\p{Alnum}	An alphanumeric character:[\p{Alpha}\p{Digit}]
\p{Punct}	Punctuation: One of !"#$%&'()*+,-./:;<=>?@[]^_`{}~
\p{Graph}	A visible character: [\p{Alnum}\p{Punct}]
\p{Print}	A printable character: [\p{Graph}\x20]
\p{Blank}	A space or a tab: [\t]
\p{Cntrl}	A control character: [\x00-\x1F\x7F]
\p{XDigit}	A hexadecimal digit: [0-9a-fA-F]
\p{Space}	A whitespace character: [\t\n\x0B\f\r]

En los caracteres imprimibles (\p{Print}), \x20 es el carácter espacio.

```
package tema3_2_ExpresionesRegulares;

public class PosixCharacterClasses {
```

```
    public void show() {

        System.out.println("a".matches("\\p{Lower}"));//true
        System.out.println("A".matches("\\p{Lower}"));//false

        System.out.println("A".matches("\\p{Upper}"));//true
        System.out.println("a".matches("\\p{Upper}"));//false
        System.out.println("aC".matches("\\p{Lower}\\p{Upper}"));//true

        System.out.println("a".matches("\\p{Alpha}"));//true
        System.out.println("A".matches("\\p{Alpha}"));//true
        System.out.println("8".matches("\\p{Alpha}"));//false

        System.out.println("8".matches("\\p{Digit}"));//true
        System.out.println("A".matches("\\p{Digit}"));//false
        System.out.println("8A".matches("\\p{Digit}\\p{Alpha}"));//true

        System.out.println("A".matches("\\p{Alnum}"));//true
        System.out.println("8".matches("\\p{Alnum}"));//true
        System.out.println("a".matches("\\p{Alnum}"));//true
        System.out.println("\n".matches("\\p{Alnum}"));//false

        System.out.println("?".matches("\\p{Punct}"));//true
        System.out.println("!".matches("\\p{Punct}"));//true
        System.out.println(";".matches("\\p{Punct}"));//true
        System.out.println("a".matches("\\p{Punct}"));//false

        System.out.println(" ".matches("\\p{Blank}"));//true
        System.out.println("\t".matches("\\p{Blank}"));//true
        System.out.println("\n".matches("\\p{Blank}"));//false

        System.out.println(" ".matches("\\p{Space}"));//true
        System.out.println("\t".matches("\\p{Space}"));//true
        System.out.println("\n".matches("\\p{Space}"));//true
        System.out.println("a".matches("\\p{Space}"));//false

    }

    public static void main(String[] args) {

        new PosixCharacterClasses().show();

    }

}
```

- **Boundary matchers**

Construct	Matches
^	The beginning of a line
$	The end of a line
\b	A word boundary

Construct	Matches
\B	A non-word boundary

```
package tema3_2_ExpresionesRegulares;

public class BoundaryMatchers {

    public void show() {

        System.out.println("hola".matches("^hola"));//true
        System.out.println("hola9".matches("^hola"));//false
        System.out.println("hola9".matches("^hola\\d"));//true
        System.out.println("ab".matches("^[aA]\\p{Lower}"));//true
        System.out.println("bA".matches("^[aA]\\p{Upper}"));//false

        System.out.println("hhola".matches(".hola$"));//true
        System.out.println("hola".matches(".hola$"));//false
        System.out.println("9hola".matches("\\dhola$"));//true
        System.out.println("hc".matches(".[abc]$"));//true
        System.out.println("ch".matches(".[abc]$"));//false

        /*
         * La expresión regular \b se llama límite de palabra
         * ya que busca en los límites de una palabra, al
         * principio o al final:
         */
        System.out.println("hola".matches("\\bhol."));//true, hol está al
principio de una palabra
        System.out.println("hola".matches(".ola\\b"));//true, ola está al final
de una palabra
        /*
         * Para realizar una búsqueda de palabras específicas
         * se coloca la palabra entre dos límites de palabra:
         */
        System.out.println("hola".matches("\\bhola\\b"));//true, hola está al
principio y al final

        // \B es justo lo contrario que \b
        System.out.println("abc".matches(".\\Bb\\B."));//true, b no está al
principio ni al final
        System.out.println("abc".matches("a\\B.."));//true, a no está al final

    }

    public static void main(String[] args) {

        new BoundaryMatchers().show();

    }

}
```

2. La clase Pattern

En Java disponemos de las clases Pattern y Matcher para poder hacer uso de las expresiones regulares. Ambas se encuentran en el paquete `java.util.regex`.

La clase **Pattern** nos permite definir el patrón, es decir, representa a la expresión regular. Veamos algunos métodos de esta clase:

- **compile**: crea un patrón a partir de una expresión regular.
- **pattern**: devuelve la expresión regular a partir de la cual se creó el patrón.
- **matches**: indica si la cadena coincide o no con la expresión regular.

```
package tema3_2_ExpresionesRegulares;

import java.util.regex.Pattern;

public class PatternClass {

    public void show() {

        Pattern pattern = Pattern.compile("\\p{Upper}\\p{Lower}");
        System.out.println(pattern.pattern());//\p{Upper}\p{Lower}

        System.out.println(Pattern.matches("\\p{Upper}\\p{Lower}", "Ho"));//true
        System.out.println(Pattern.matches("\\p{Upper}\\p{Lower}", "ho"));//false

    }

    public static void main(String[] args) {

        new PatternClass().show();

    }

}
```

3. La clase Matcher

La clase **Matcher** realiza operaciones de coincidencia del patrón en una secuencia de caracteres.

Se puede crear un objeto de tipo Matcher mediante el método ***matcher*** de la clase Pattern. Una vez creado, un matcher puede ser utilizado para realizar tres tipos diferentes de operaciones:

- **matches**: intenta hacer coincidir toda la secuencia de entrada con el patrón.
- **lookingAt**: intenta hacer coincidir el principio de la secuencia de entrada con el patrón.
- **find**: intenta encontrar la próxima secuencia de entrada que coincide con el patrón. Si hay varias coincidencias con el patrón dentro del mismo texto, cada llamada a find devolverá la siguiente coincidencia.
- **start**: devuelve el índice de la secuencia de entrada donde empieza la coincidencia del último find.
- **end**: devuelve el índice de la secuencia de entrada del carácter que está justo después del último de la coincidencia del último find.

- **reset**: resetea el matcher.

```
package tema3_2_ExpresionesRegulares;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class MatcherClass {

    public void show() {

        int count;
        //Con Pattern.CASE_INSENSITIVE, no se distingue entre mayúsculas y
minúsculas
        Pattern pattern = Pattern.compile("es", Pattern.CASE_INSENSITIVE);
        Matcher matcher = pattern.matcher("Estoy en España");
        System.out.println(matcher.matches());//false
        System.out.println(matcher.lookingAt());//true

        matcher.reset("Esto es un escrito en español");
        count = 0;
        while (matcher.find()) {
            count++;
            System.out.printf("Coincidencia número %d: empieza en %d y termina en
%d\n", count, matcher.start(),
                    matcher.end() - 1);
        }

    }

    public static void main(String[] args) {

        new MatcherClass().show();

    }

}
```

La expresión regular . coincide con cualquier carácter excepto un terminador de línea, a menos que se especifique la bandera DOTALL.

```
package tema3_2_ExpresionesRegulares;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Point {

    public void show() {

        Pattern pattern = Pattern.compile(".", Pattern.DOTALL);
        Matcher matcher = pattern.matcher("\n");
        System.out.println(matcher.matches());//true

        System.out.println("\n".matches("."));//false
```

```
    }

    public static void main(String[] args) {

        new Point().show();

    }

}
```

Veamos más ejemplos de *boundary matchers* utilizando objetos de tipo Matcher:

```
package tema3_2_ExpresionesRegulares;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class BoundaryMatchers2 {

    public void show() {

        String text = "Esto es un texto escrito en español";
        Pattern pattern;
        Matcher matcher;

        pattern = Pattern.compile("\\be", Pattern.CASE_INSENSITIVE);
        matcher = pattern.matcher(text);
        System.out.println("Las palabras que empiezan por e o por E se encuentran
en las siguientes posiciones:");
        while (matcher.find()) {
            System.out.println(matcher.start());
        }

        pattern = Pattern.compile("\\Be");
        matcher = pattern.matcher(text);
        System.out.println("Las e que no son comienzos de palabra se encuentran
en las siguientes posiciones:");
        while (matcher.find()) {
            System.out.println(matcher.start());
        }

        pattern = Pattern.compile("o\\b");
        matcher = pattern.matcher(text);
        System.out.println("Las o que son finales de palabras se encuentran en
las siguientes posiciones:");
        while (matcher.find()) {
            System.out.println(matcher.start());
        }

        pattern = Pattern.compile("o\\B");
        matcher = pattern.matcher(text);
        System.out.println("Las o que no son finales de palabras se encuentran en
las siguientes posiciones:");
        while (matcher.find()) {
```

```
            System.out.println(matcher.start());
        }
        /*
         * Para realizar una búsqueda de palabras específicas
         * se coloca la palabra entre dos límites de palabra:
         */
        pattern = Pattern.compile("\\btexto\\b");
        matcher = pattern.matcher(text);
        System.out.println("La palabra texto se encuentra en las siguientes
posiciones:");
        while (matcher.find()) {
            System.out.println(matcher.start());
        }

        pattern = Pattern.compile("^Lín.*", Pattern.MULTILINE);
        matcher = pattern.matcher("""
                                  Línea 1
                                  Línea 2
                                  Línea 3
                                  """);
        System.out.println("Las líneas que empiezan por Lín se encuentran en las
siguientes posiciones:");
        while (matcher.find()) {
            System.out.println(matcher.start());
        }

        pattern = Pattern.compile(".*nea$", Pattern.MULTILINE);
        matcher = pattern.matcher("""
                                  Primera Línea
                                  Segunda Línea
                                  Tercera Línea
                                  """);
        System.out.println("Las líneas que finalizan por nea se encuentran en las
siguientes posiciones:");
        while (matcher.find()) {
            System.out.println(matcher.start());
        }

    }

    public static void main(String[] args) {

        new BoundaryMatchers2().show();

    }

}
```

En el ejemplo, la bandera **Pattern.MULTILINE** permite que `^` coincida con el inicio de cada línea, no solo con el inicio de toda la cadena. Por el mismo motivo, permite que `$` coincida con el final de cada línea y no solo con el final de toda la cadena.

4. Grupos

Los grupos sirven para extraer partes de una cadena y se marcan con un paréntesis en la expresión regular. Cuando se encuentra una coincidencia en un texto, se puede acceder a la parte que se encuentra dentro del grupo a través del método ***group***. Una expresión regular puede tener más de un grupo, en cuyo caso cada uno lleva sus propios paréntesis. El grupo con número 0 es toda la expresión regular y los grupos marcados con paréntesis empiezan a numerarse a partir del 1. También puede haber grupos dentro de otros grupos, como en el siguiente ejemplo, donde el grupo 2 y el grupo 3 están dentro del grupo 1.

```
package tema3_2_ExpresionesRegulares;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class Group {

    public void show() {

        //Como veremos en el siguiente apartado,el + es 1 ó más veces
        Pattern pattern = Pattern.compile("((\\d+)\\+(\\d+))=(\\d+)");
        Matcher matcher = pattern.matcher("23+56=79;15+13=28;30+60=90");

        while (matcher.find()) {
            System.out.println(matcher.group(0));
            System.out.printf("Sumandos: %s\n", matcher.group(1));
            System.out.printf("Sumando 1: %s\n", matcher.group(2));
            System.out.printf("Sumando 2: %s\n", matcher.group(3));
            System.out.printf("Resultado: %s\n\n", matcher.group(4));

        }

    }

    public static void main(String[] args) {

        new Group().show();

    }

}
```

5. Quantifiers (cuantificadores)

- **Greedy quantifiers**

Por defecto, los cuantificadores son greedy. Se llaman *greedy* (glotón) porque tratan de coger lo máximo posible de la cadena pero siempre intentando que el patrón completo se cumpla.

Construct	Matches
X?	X, once or not at all
X*	X, zero or more times

Construct	Matches
X+	X, one or more times
X{n}	X, exactly n times
X{n,}	X, at least n times
X{n,m}	X, at least n but not more than m times

```java
package tema3_2_ExpresionesRegulares;

public class GreedyQuantifiers {

    public void show() {

        /*
         * El símbolo ? sirve para indicar que la expresión que le
         * antecede puede aparecer una o ninguna vez
         */
        System.out.println("a".matches("a?"));//true
        System.out.println("".matches("a?"));//true
        System.out.println("b".matches("a?"));//false

        /*
         * El símbolo * indica que la expresión puede repetirse
         * ninguna o varias veces
         */
        System.out.println("a".matches("a*"));//true
        System.out.println("".matches("a*"));//true
        System.out.println("aaaa".matches("a*"));//true
        System.out.println("b".matches("a*"));//false

        /*
         * El símbolo + indica que la expresión puede repetirse
         * una o varias veces
         */
        System.out.println("a".matches("a+"));//true
        System.out.println("aaaa".matches("a+"));//true
        System.out.println("".matches("a+"));//false
        System.out.println("b".matches("a+"));//false

        //Un número entre llaves indica las veces que se repite la expresión
        System.out.println("aaa".matches("a{3}"));//true
        System.out.println("aaa".matches("a{4}"));//false

        /*
         * Un número entre llaves con una coma indica el mínimo
         * de veces que se repite la expresión
         */
        System.out.println("aaa".matches("a{3,}"));//true
        System.out.println("aaaaa".matches("a{3,}"));//true
        System.out.println("aa".matches("a{3,}"));//false

        /*
```

```
         * Como el anterior pero aparece un segundo número que
         * indica el máximo de veces que se repite la expresión
         */
        System.out.println("aaa".matches("a{3,6}"));//true
        System.out.println("aaaaaa".matches("a{3,6}"));//true
        System.out.println("aa".matches("a{3,6}"));//false
        System.out.println("aaaaaaa".matches("a{3,6}"));//false

    }

    public static void main(String[] args) {

        new GreedyQuantifiers().show();

    }

}
```

- **Reluctant quantifiers**

Otro posible comportamiento es *reluctant* (reacio, reticente). Este comportamiento es el contrario de greedy, trata de coger lo menos posible pero siempre intentando que se cumpla el patrón.

Se escriben como los greedy pero añadiendo un ? detrás.

Construct	Matches
X??	X, once or not at all
X*?	X, zero or more times
X+?	X, one or more times
X{n}?	X, exactly n times
X{n,}?	X, at least n times
X{n,m}?	X, at least n but not more than m times

En el siguiente ejemplo se puede apreciar la diferencia entre greedy y reluctant:

```
package tema3_2_ExpresionesRegulares;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ReluctantQuantifiers {

    public void show() {

        Pattern patternGreedy = Pattern.compile("a+");
        Pattern patternReluctant = Pattern.compile("a+?");
        Matcher matcherGreedy = patternGreedy.matcher("aaaa");
        Matcher matcherReluctant = patternReluctant.matcher("aaaa");

        while (matcherGreedy.find()) {
```

```
            System.out.printf("Greedy: coincidencia desde %d hasta %d\n",
matcherGreedy.start(),
                    matcherGreedy.end() - 1);
        }
        while (matcherReluctant.find()) {
            System.out.printf("Reluctant: coincidencia desde %d hasta %d\n",
matcherReluctant.start(),
                    matcherReluctant.end() - 1);
        }

    }

    public static void main(String[] args) {

        new ReluctantQuantifiers().show();

    }

}
```

La salida por consola es la siguiente:

```
Greedy: coincidencia desde 0 hasta 3
Reluctant: coincidencia desde 0 hasta 0
Reluctant: coincidencia desde 1 hasta 1
Reluctant: coincidencia desde 2 hasta 2
Reluctant: coincidencia desde 3 hasta 3
```

- **Possessive quantifiers**

Tratan de coger lo máximo posible de la cadena pero no se preocupan de que se cumpla el patrón. Este comportamiento se usa únicamente por motivos de eficiencia. Si en la cadena hay un trozo que queramos quitar y que podamos distinguir con una expresión regular, podemos ponerlo con este modo possesive. De esta forma, el possesive se comerá directamente ese trozo de cadena y no perderá el tiempo tratando de hacer casar ese trozo con el patrón de alguna u otra forma.

Se escriben como los greedy pero añadiendo un + detrás.

Construct	Matches
X?+	X, once or not at all
X*+	X, zero or more times
X++	X, one or more times
X{n}+	X, exactly n times
X{n,}+	X, at least n times
X{n,m}+	X, at least n but not more than m times

En el siguiente ejemplo se puede apreciar la diferencia entre greedy y possessive:

```
package tema3_2_ExpresionesRegulares;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class PossessiveQuantifiers {

    public void show() {

        Pattern patternGreedy = Pattern.compile("a+a");
        Pattern patternPossessive = Pattern.compile("a++a");
        Matcher matcherGreedy = patternGreedy.matcher("aaaa");
        Matcher matcherPossessive = patternPossessive.matcher("aaaa");

        while (matcherGreedy.find()) {
            System.out.printf("Greedy: coincidencia desde %d hasta %d\n",
matcherGreedy.start(),
                    matcherGreedy.end() - 1);
        }
        while (matcherPossessive.find()) {
            System.out.printf("Possessive: coincidencia desde %d hasta %d\n",
matcherPossessive.start(),
                    matcherPossessive.end() - 1);
        }

    }

    public static void main(String[] args) {

        new PossessiveQuantifiers().show();

    }

}
```

La salida por consola es la siguiente:

```
Greedy: coincidencia desde 0 hasta 3
```

En el modo possessive, con la primera parte del patrón `a++` ya coge la cadena completa `"aaaa"`. Como no se preocupa de que el patrón se cumpla, entonces la última `a` del patrón ya no tiene coincidencia, con lo que el find devuelve false.

6. Reemplazar partes de una cadena

La clase Matcher tiene métodos para reemplazar partes de la cadena que cumplan con el patrón:

- **replaceAll**: reemplaza todas las coincidencias.
- **replaceFirst**: reemplaza solamente la primera coincidencia.

```
package tema3_2_ExpresionesRegulares;

import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```
public class Replace {

    public void show() {

        Pattern pattern = Pattern.compile("\\d");
        Matcher matcher = pattern.matcher("Tengo 20 años y vivo en la calle
Puerto Real 15");

        String replaceAll = matcher.replaceAll("*");
        String replaceFirst = matcher.replaceFirst("*");

        System.out.printf("replaceAll: %s\n", replaceAll);
        System.out.printf("replaceFirst: %s\n", replaceFirst);

    }

    public static void main(String[] args) {

        new Replace().show();

    }

}
```

Salida por consola:

```
replaceAll: Tengo ** años y vivo en la calle Puerto Real **
replaceFirst: Tengo *0 años y vivo en la calle Puerto Real 15
```