The java.util.regex.Matcher class is used to search through a text for multiple occurrences of a regular expression. You can also use a Matcher to search for the same regular expression in different texts.

The Matcher class has a lot of useful methods. For a full list, see the official JavaDoc for the Matcher class. I will cover the core methods here. Here is a list of the methods covered:

* [**Creating a Matcher**](http://tutorials.jenkov.com/java-regex/matcher.html#0)
* [**matches()**](http://tutorials.jenkov.com/java-regex/matcher.html#1)
* [**lookingAt()**](http://tutorials.jenkov.com/java-regex/matcher.html#2)
* [**find() + start() + end()**](http://tutorials.jenkov.com/java-regex/matcher.html#3)
* [**reset()**](http://tutorials.jenkov.com/java-regex/matcher.html#4)
* [**group()**](http://tutorials.jenkov.com/java-regex/matcher.html#5)
  + [**Multiple Groups**](http://tutorials.jenkov.com/java-regex/matcher.html#6)
  + [**Groups Inside Groups**](http://tutorials.jenkov.com/java-regex/matcher.html#7)
* [**replaceAll() + replaceFirst()**](http://tutorials.jenkov.com/java-regex/matcher.html#8)
* [**appendReplacement() + appendTail()**](http://tutorials.jenkov.com/java-regex/matcher.html#9)

**Creating a Matcher**

Creating a Matcher is done via the matcher() method in the Pattern class. Here is an example:

String text =

"This is the text to be searched " +

"for occurrences of the http:// pattern.";

String patternString = ".\*http://.\*";

Pattern pattern = Pattern.compile(patternString);

**Matcher matcher = pattern.matcher(text);**

**matches()**

The matches() method in the Matcher class matches the regular expression against the whole text passed to the Pattern.matcher() method, when the Matcher was created. Here is an example:

boolean matches = matcher.matches();

If the regular expression matches the whole text, then the matches() method returns true. If not, the matches() method returns false.

You cannot use the matches() method to search for multiple occurrences of a regular expression in a text. For that, you need to use the find(), start() and end() methods.

**lookingAt()**

The lookingAt() method works like the matches() method with one major difference. The lookingAt() method only matches the regular expression against the beginning of the text, whereas matches() matches the regular expression against the whole text. In other words, if the regular expression matches the beginning of a text but not the whole text, lookingAt() will return true, whereas matches() will return false.

Here is an example:

String text =

"This is the text to be searched " +

"for occurrences of the http:// pattern.";

String patternString = "This is the";

Pattern pattern = Pattern.compile(patternString, Pattern.CASE\_INSENSITIVE);

Matcher matcher = pattern.matcher(text);

System.out.println("lookingAt = " + matcher.lookingAt());

System.out.println("matches = " + matcher.matches());

This example matches the regular expression "this is the" against both the beginning of the text, and against the whole text. Matching the regular expression against the beginning of the text (lookingAt()) will return true.

Matching the regular expression against the whole text (matches()) will return false, because the text has more characters than the regular expression. The regular expression says that the text must match the text "This is the" exactly, with no extra characters before or after the expression.

**find() + start() + end()**

The find() method searches for occurrences of the regular expressions in the text passed to the Pattern.matcher(text) method, when the Matcher was created. If multiple matches can be found in the text, the find() method will find the first, and then for each subsequent call to find() it will move to the next match.

The methods start() and end() will give the indexes into the text where the found match starts and ends.

Here is an example:

String text =

"This is the text which is to be searched " +

"for occurrences of the word 'is'.";

String patternString = "is";

Pattern pattern = Pattern.compile(patternString);

Matcher matcher = pattern.matcher(text);

int count = 0;

while(matcher.find()) {

count++;

System.out.println("found: " + count + " : "

+ matcher.start() + " - " + matcher.end());

}

This example will find the pattern "is" four times in the searched string. The output printed will be this:

found: 1 : 2 - 4

found: 2 : 5 - 7

found: 3 : 23 - 25

found: 4 : 70 - 72

**reset()**

The reset() method resets the matching state internally in the Matcher. In case you have started matching occurrences in a string via the find() method, the Matcher will internally keep a state about how far it has searched through the input text. By calling reset() the matching will start from the beginning of the text again.

There is also a reset(CharSequence) method. This method resets the Matcher, and makes the Matcher search through the CharSequence passed as parameter, instead of the CharSequence the Matcher was originally created with.

**group()**

Imagine you are searching through a text for URL's, and you would like to extract the found URL's out of the text. Of course you could do this with the start() and end() methods, but it is easier to do so with the group functions.

Groups are marked with parentheses in the regular expression. For instance:

(John)

This regular expression matches the text John. The parentheses are not part of the text that is matched. The parentheses mark a group. When a match is found in a text, you can get access to the part of the regular expression inside the group.

You access a group using the group(int groupNo) method. A regular expression can have more than one group. Each group is thus marked with a separate set of parentheses. To get access to the text that matched the subpart of the expression in a specific group, pass the number of the group to the group(int groupNo) method.

The group with number 0 is always the whole regular expression. To get access to a group marked by parentheses you should start with group numbers 1.

Here is an example:

String text =

"John writes about this, and John writes about that," +

" and John writes about everything. "

;

String patternString1 = "(John)";

Pattern pattern = Pattern.compile(patternString1);

Matcher matcher = pattern.matcher(text);

while(matcher.find()) {

System.out.println("found: " + matcher.group(1));

}

This example searches the text for occurrences of the word John. For each match found, group number 1 is extracted, which is what matched the group marked with parentheses. The output of the example is:

found: John

found: John

found: John

**Multiple Groups**

As mentioned earlier, a regular expression can have multiple groups. Here is a regular expression illustrating that:

(John) (.+?)

This expression matches the text "John" followed by a space, and then one or more characters. You cannot see it in the example above, but there is a space after the last group too.

This expression contains a few characters with special meanings in a regular expression. The . means "any character". The + means "one or more times", and relates to the . (any character, one or more times). The ? means "match as small a number of characters as possible".

Here is a full code

String text =

"John writes about this, and John Doe writes about that," +

" and John Wayne writes about everything."

;

String patternString1 = "(John) (.+?) ";

Pattern pattern = Pattern.compile(patternString1);

Matcher matcher = pattern.matcher(text);

while(matcher.find()) {

System.out.println("found: " + **matcher.group(1)** +

" " + **matcher.group(2))**;

}

Notice the reference to the two groups, marked in bold. The characters matched by those groups are printed to System.out. Here is what the example prints out:

found: John writes

found: John Doe

found: John Wayne

**Groups Inside Groups**

It is possible to have groups inside groups in a regular expression. Here is an example:

((John) (.+?))

Notice how the two groups from the examples earlier are now nested inside a larger group. (again, you cannot see the space at the end of the expression, but it is there).

When groups are nested inside each other, they are numbered based on when the left paranthesis of the group is met. Thus, group 1 is the big group. Group 2 is the group with the expression John inside. Group 3 is the group with the expression .+? inside. This is important to know when you need to reference the groups via the groups(int groupNo) method.

Here is an example that uses the above nested groups:

String text =

"John writes about this, and John Doe writes about that," +

" and John Wayne writes about everything."

;

String patternString1 = "((John) (.+?)) ";

Pattern pattern = Pattern.compile(patternString1);

Matcher matcher = pattern.matcher(text);

while(matcher.find()) {

System.out.println("found: <" + matcher.group(1) +

"> <" + matcher.group(2) +

"> <" + matcher.group(3) + ">");

}

Here is the output from the above example:

found: <John writes> <John> <writes>

found: <John Doe> <John> <Doe>

found: <John Wayne> <John> <Wayne>

Notice how the value matched by the first group (the outer group) contains the values matched by both of the inner groups.

**replaceAll() + replaceFirst()**

The replaceAll() and replaceFirst() methods can be used to replace parts of the string the Matcher is searching through. The replaceAll() method replaces all matches of the regular expression. The replaceFirst() only replaces the first match.

Before any matching is carried out, the Matcher is reset, so that matching starts from the beginning of the input text.

Here are two examples:

String text =

"John writes about this, and John Doe writes about that," +

" and John Wayne writes about everything."

;

String patternString1 = "((John) (.+?)) ";

Pattern pattern = Pattern.compile(patternString1);

Matcher matcher = pattern.matcher(text);

String replaceAll = matcher.replaceAll("Joe Blocks ");

System.out.println("replaceAll = " + replaceAll);

String replaceFirst = matcher.replaceFirst("Joe Blocks ");

System.out.println("replaceFirst = " + replaceFirst);

And here is what the example outputs:

replaceAll = Joe Blocks about this, and Joe Blocks writes about that,

and Joe Blocks writes about everything.

replaceFirst = Joe Blocks about this, and John Doe writes about that,

and John Wayne writes about everything.

The line breaks and indendation of the following line is not really part of the output. I added them to make the output easier to read.

Notice how the first string printed has all occurrences of John with a word after, replaced with the string Joe Blocks. The second string only has the first occurrence replaced.

**appendReplacement() + appendTail()**

The appendReplacement() and appendTail() methods are used to replace string tokens in an input text, and append the resulting string to a StringBuffer.

When you have found a match using the find() method, you can call the appendReplacement(). Doing so results in the characters from the input text being appended to the StringBuffer, and the matched text being replaced. Only the characters starting from then end of the last match, and until just before the matched characters are copied.

The appendReplacement() method keeps track of what has been copied into the StringBuffer, so you can continue searching for matches using find() until no more matches are found in the input text.

Once the last match has been found, a part of the input text will still not have been copied into the StringBuffer. This is the characters from the end of the last match and until the end of the input text. By calling appendTail() you can append these last characters to the StringBuffer too.

Here is an example:

String text =

"John writes about this, and John Doe writes about that," +

" and John Wayne writes about everything."

;

String patternString1 = "((John) (.+?)) ";

Pattern pattern = Pattern.compile(patternString1);

Matcher matcher = pattern.matcher(text);

StringBuffer stringBuffer = new StringBuffer();

while(matcher.find()){

matcher.appendReplacement(stringBuffer, "Joe Blocks ");

System.out.println(stringBuffer.toString());

}

matcher.appendTail(stringBuffer);

System.out.println(stringBuffer.toString());

Notice how appendReplacement() is called inside the while(matcher.find()) loop, and appendTail() is called just after the loop.

The output from this example is:

Joe Blocks

Joe Blocks about this, and Joe Blocks

Joe Blocks about this, and Joe Blocks writes about that, and Joe Blocks

Joe Blocks about this, and Joe Blocks writes about that, and Joe Blocks

writes about everything.

The line break in the last line is inserted by me, to make the text more readable. In the real output there would be no line break.

As you can see, the StringBuffer is built up by characters and replacements from the input text, one match at a time.