What Are Regular Expressions?

A *regular expression*  a pattern for a String. Regular Expressions can be used to search, edit or manipulate text. Regular expressions are not language specific but they differ slightly for each language. Java regular expressions are most similar to Perl.

Java Regular Expression classes are present in ***java.util.regex*** package that contains three classes: **Pattern**,**Matcher** and **PatternSyntaxException**.

1. Pattern object is the compiled version of the regular expression. It doesn’t have any public constructor and we use it’s public static method ***compile*** to create the pattern object by passing regular expression argument.

2. Matcher is the regex engine object that matches the input String pattern with the pattern object created. This class doesn’t have any public construtor and we get a Matcher object using pattern object ***matcher***method that takes the input String as argument. We then use ***matches*** method that returns boolean result based on input String matches the regex pattern or not.

3. PatternSyntaxException is thrown if the regular expression syntax is not correct.

Let’s see all these classes in action with a simple example:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | package com.journaldev.util;    import java.util.regex.\*;    public class PatternExample {        public static void main(String[] args) {          Pattern pattern = Pattern.compile(".xx.");          Matcher matcher = pattern.matcher("MxxY");          System.out.println("Input String matches regex - "+matcher.matches());          // bad regular expression          pattern = Pattern.compile("\*xx\*");        }    } |

Output of the above program is:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | Input String matches regex - true  Exception in thread "main" java.util.regex.PatternSyntaxException: Dangling meta character '\*' near index 0  \*xx\*  ^      at java.util.regex.Pattern.error(Pattern.java:1924)      at java.util.regex.Pattern.sequence(Pattern.java:2090)      at java.util.regex.Pattern.expr(Pattern.java:1964)      at java.util.regex.Pattern.compile(Pattern.java:1665)      at java.util.regex.Pattern.(Pattern.java:1337)      at java.util.regex.Pattern.compile(Pattern.java:1022)      at com.journaldev.util.PatternExample.main(PatternExample.java:13) |

Since regular expressions are revolved around String, String class has been extended in Java 1.4 to provide a ***matches*** method that does regex pattern matching. Internally it uses Pattern and Matcher classes to do the processing but obviously it reduces the code lines.

Pattern class also contains ***matches*** method that takes regex and input String as argument and return boolean result after matching them.

So below code works fine for matching input String with regular expression.

|  |  |
| --- | --- |
| 1  2  3 | String str = "bbb";  System.out.println("Using String matches method: "+str.matches(".bb"));  System.out.println("Using Pattern matches method: "+Pattern.matches(".bb", str)); |

So if your requirement is just to check if the input String matches with the pattern, you should save time and lines of code by using simple String matches method.

You should use Pattern and Matches classes only when you need to manipulate the input String or you need to reuse the pattern.

Note that the pattern defined by regex is applied on the String from left to right and once a source character is used in a match, it can’t be reused.

For example, regex “121” will match “31212142121” only twice as “\_121\_\_\_\_121″.

Regular Expressions common matching symbols

| **Regular Expression** | **Description** | **Example** |
| --- | --- | --- |
| . | Matches any single character | (“..”, “a%”) – true(“..”, “.a”) – true  (“..”, “a”) – false |
| ^xxx | Matches xxx regex at the beginning of the line | (“^a.c.”, “abcd”) – true  (“^a”, “ac”) – false |
| xxx$ | Matches regex xxx at the end of the line | (“..cd$”, “abcd”) – true(“a$”, “a”) – true  (“a$”, “aca”) – false |
| [abc] | Can match any of the letter a, b or c. [] are known as character classes. | (“^[abc]d.”, “ad9″) – true  (“[ab].d$”, “bad”) – true  (“[ab]x”, “cx”) – false |
| [abc][12] | Can match a, b or c followed by 1 or 2 | (“[ab][12].”, “a2#”) – true  (“[ab]..[12]“, “acd2″) – true  (“[ab][12]“, “c2″) – false |
| [^abc] | When ^ is the first character in [], it negates the pattern, matches anything except a, b or c | (“[^ab][^12].”, “c3#”) – true  (“[^ab]..[^12]“, “xcd3″) – true  (“[^ab][^12]“, “c2″) – false |
| [a-e1-8] | Matches ranges between a to e or 1 to 8 | (“[a-e1-3].”, “d#”) – true  (“[a-e1-3]“, “2”) – true  (“[a-e1-3]“, “f2″) – false |
| xx|yy | Matches regex xx or yy | (“x.|y”, “xa”) – true(“x.|y”, “y”) – true  (“x.|y”, “yz”) – false |

Java Regular Expressions Metacharacters

We have some metacharacters also in regular expression, it’s like shortcodes for common matching patterns.

|  |  |
| --- | --- |
| **Regular Expression** | **Description** |
| \d | Any digits, short of [0-9] |
| \D | Any non-digit, short for [^0-9] |
| \s | Any whitespace character, short for [\t\n\x0B\f\r] |
| \S | Any non-whitespace character, short for [^\s] |
| \w | Any word character, short for [a-zA-Z\_0-9] |
| \W | Any non-word character, short for [^\w] |
| \b | A word boundary |
| \B | A non word boundary |

There are two ways to use metacharacters as ordinary characters in regular expressions.

1. Precede the metacharacter with a backslash (\).
2. Keep metacharcter within \Q (which starts the quote) and \E (which ends it).

Regular Expression Quantifiers

Quantifiers specify the number of occurrence of a character to match against.

|  |  |
| --- | --- |
| **Regular Expression** | **Description** |
| x? | x occurs once or not at all |
| X\* | X occurs zero or more times |
| X+ | X occurs one or more times |
| X{n} | X occurs exactly n times |
| X{n,} | X occurs n or more times |
| X{n,m} | X occurs at least n times but not more than m times |

Quantifiers can be used with character classes and capturing groups also.

For example, [abc]+ means a, b or c one or more times.

(abc)+ means the group “abc” one more more times. We will discuss about **Capturing Group** now.

Regular Expression Capturing Groups

Capturing groups are used to treat multiple characters as a single unit.  You can create a group using **()**. The portion of input String that matches the capturing group is saved into memory and can be recalled using ***Backreference***.

You can use ***matcher.groupCount***method to find out the number of capturing groups in a regex pattern. For example in ((a)(bc)) contains 3 capturing groups; ((a)(bc)), (a) and (bc) .

You can use ***Backreference*** in regular expression with backslash (\) and then the number of group to be recalled.

Capturing groups and Backreferences can be confusing, so let’s understand this with an example.

|  |  |
| --- | --- |
| 1  2  3  4 | System.out.println(Pattern.matches("(\\w\\d)\\1", "a2a2")); //true  System.out.println(Pattern.matches("(\\w\\d)\\1", "a2b2")); //false  System.out.println(Pattern.matches("(AB)(B\\d)\\2\\1", "ABB2B2AB")); //true  System.out.println(Pattern.matches("(AB)(B\\d)\\2\\1", "ABB2B3AB")); //false |

In the first example, at **runtime** first capturing group is (\w\d) which evaluates to “a2″ when matched with the input String “a2a2″ and saved in memory. So \1 is referring to “a2″ and hence it returns true. Due to same reason second statement prints false.  
Try to understand this scenario for statement 3 and 4 yourself. :)

Now we will look at some important methods of Pattern and Matcher classes.

We can create a Pattern object with flags. For example **Pattern.*CASE\_INSENSITIVE*** enables case insensitive matching.

Pattern class also provides ***split(String)*** that is similar to String class *split()* method.  
Pattern class ***toString()*** method returns the regular expression String from which this pattern was compiled.

Matcher classes have ***start()*** and ***end()*** index methods that show precisely where the match was found in the input string.

Matcher class also provides String manipulation methods ***replaceAll(String replacement)***and ***replaceFirst(String replacement)***.

Now we will see these common functions in action through a simple java class:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | package com.journaldev.util;    import java.util.regex.Matcher;  import java.util.regex.Pattern;    public class RegexExamples {        public static void main(String[] args) {          // using pattern with flags          Pattern pattern = Pattern.compile("ab", Pattern.CASE\_INSENSITIVE);          Matcher matcher = pattern.matcher("ABcabdAb");          // using Matcher find(), group(), start() and end() methods          while (matcher.find()) {              System.out.println("Found the text \"" + matcher.group()                      + "\" starting at " + matcher.start()                      + " index and ending at index " + matcher.end());          }            // using Pattern split() method          pattern = Pattern.compile("\\W");          String[] words = pattern.split("one@two#three:four$five");          for (String s : words) {              System.out.println("Split using Pattern.split(): " + s);          }            // using Matcher.replaceFirst() and replaceAll() methods          pattern = Pattern.compile("1\*2");          matcher = pattern.matcher("11234512678");          System.out.println("Using replaceAll: " + matcher.replaceAll("\_"));          System.out.println("Using replaceFirst: " + matcher.replaceFirst("\_"));      }    } |

Output of the above program is:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | Found the text "AB" starting at 0 index and ending at index 2  Found the text "ab" starting at 3 index and ending at index 5  Found the text "Ab" starting at 6 index and ending at index 8  Split using Pattern.split(): one  Split using Pattern.split(): two  Split using Pattern.split(): three  Split using Pattern.split(): four  Split using Pattern.split(): five  Using replaceAll: \_345\_678  Using replaceFirst: \_34512678 |

Regular expressions are one of the area of [java interview questions](http://www.journaldev.com/tag/java-interview-questions) and in next few posts, I will provide some real life examples.