**Chapter 7 : Regular Expression**

**What is a Regular Expression**?

A regular expression is a sequence of characters that forms a search pattern. When you search for data in a text, you can use this search pattern to describe what you are searching for.

A regular expression can be a single character, or a more complicated pattern.

Regular expressions can be used to perform all types of text search and text replace operations.

Java does not have a built-in Regular Expression class, but we can import the ***java.util.regex*** package to work with regular expressions. The package includes the following classes:

* **MatchResult** interface
* **Matcher** class : Used to search for the pattern
* **Pattern** class : Defines a pattern (to be used in a search)
* **PatternSyntaxException** class : Indicates syntax error in a regular expression pattern



https://www.tutorialspoint.com/java/java\_methods.htm

https://www.w3schools.com/java/java\_methods.asp

https://www.geeksforgeeks.org/methods-in-java/

## Matcher class

It implements the **MatchResult** interface. It is a regex engine which is used to perform match operations on a character sequence.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | boolean matches() | test whether the regular expression matches the pattern. |
| 2 | boolean find() | finds the next expression that matches the pattern. |
| 3 | boolean find(int start) | finds the next expression that matches the pattern from the given start number. |
| 4 | String group() | returns the matched subsequence. |
| 5 | int start() | returns the starting index of the matched subsequence. |
| 6 | int end() | returns the ending index of the matched subsequence. |
| 7 | int groupCount() | returns the total number of the matched subsequence. |

## Pattern class

It is the compiled version of a regular expression. It is used to define a pattern for the regex engine.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1 | static Pattern compile(String regex) | compiles the given regex and returns the instance of the Pattern. |
| 2 | Matcher matcher(CharSequence input) | creates a matcher that matches the given input with the pattern. |
| 3 | static boolean matches(String regex, CharSequence input) | It works as the combination of compile and matcher methods. It compiles the regular expression and matches the given input with the pattern. |
| 4 | String[] split(CharSequence input) | splits the given input string around matches of given pattern. |
| 5 | String pattern() | returns the regex pattern. |

### Flags

### Flags in the compile() method change how the search is performed. Here are a few of them:

### Pattern.CASE\_INSENSITIVE - The case of letters will be ignored when performing a search.

### Pattern.LITERAL - Special characters in the pattern will not have any special meaning and will be treated as ordinary characters when performing a search.

### Pattern.UNICODE\_CASE - Use it together with the CASE\_INSENSITIVE flag to also ignore the case of letters outside of the English alphabet

### *String content = "This is a tutorial Website!";*

### *String patternString = ".\*tuToRiAl.";*

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

### PatternSyntaxException Class

### A PatternSyntaxException is an unchecked exception that indicates a syntax error in a regular expression pattern. The PatternSyntaxException class provides the following methods to help you determine what went wrong –

|  |  |
| --- | --- |
| Sr.No. | Method & Description |
| 1 | public String getDescription()Retrieves the description of the error. |
| 2 | public int getIndex()Retrieves the error index. |
| 3 | public String getPattern()Retrieves the erroneous regular expression pattern. |
| 4 | public String getMessage()Returns a multi-line string containing the description of the syntax error and its index, the erroneous regular expression pattern, and a visual indication of the error index within the pattern. |

### Example of Java Regular Expressions

There are three ways to write the regex example in Java.

**import** java.util.regex.\*;

**public** **class** RegexExample1{

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

//1st way

Pattern p = Pattern.compile(".s");//. represents single character

Matcher m = p.matcher("as");

**boolean** b = m.matches();

//2nd way

**boolean** b2=Pattern.compile(".s").matcher("as").matches();

//3rd way

**boolean** b3 = Pattern.matches(".s", "as");

System.out.println(b+" "+b2+" "+b3);

}}

#### **Output**

true true true

## Regular Expression . Example

The . (dot) represents a single character.

**import** java.util.regex.\*;

**class** RegexExample2{

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

System.out.println(Pattern.matches(".s", "as"));//true (2nd char is s)

System.out.println(Pattern.matches(".s", "mk"));//false (2nd char is not s)

System.out.println(Pattern.matches(".s", "mst"));//false (has more than 2 char)

System.out.println(Pattern.matches(".s", "amms"));//false (has more than 2 char)

System.out.println(Pattern.matches("..s", "mas"));//true (3rd char is s)

}}

## Regex Character classes

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Character Class** | **Description** | |
| 1 | [abc] | | a, b, or c (simple class) |
| 2 | [^abc] | | Any character except a, b, or c (negation) |
| 3 | [a-zA-Z] | | a through z or A through Z, inclusive (range) |
| 4 | [a-d[m-p]] | | a through d, or m through p: [a-dm-p] (union) |
| 5 | [a-z&&[def]] | | d, e, or f (intersection) |
| 6 | [a-z&&[^bc]] | | a through z, except for b and c: [ad-z] (subtraction) |
| 7 | [a-z&&[^m-p]] | | a through z, and not m through p: [a-lq-z](subtraction) |

## Regular Expression Character classes Example

**import** java.util.regex.\*;

**class** RegexExample3{

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

System.out.println(Pattern.matches("[amn]", "abcd"));//false (not a or m or n)

System.out.println(Pattern.matches("[amn]", "a"));//true (among a or m or n)

System.out.println(Pattern.matches("[amn]", "ammmna"));//false (m and a  comes more than once)

}}

## Regex Quantifiers

The quantifiers specify the number of occurrences of a character.

|  |  |
| --- | --- |
| **Regex** | **Description** |
| X? | X occurs once or not at all |
| X+ | X occurs once or more times |
| X\* | X occurs zero or more times |
| X{n} | X occurs n times only |
| X{n,} | X occurs n or more times |
| X{y,z} | X occurs at least y times but less than z times |

## Regular Expression Character classes and Quantifiers Example

**import** java.util.regex.\*;

**class** RegexExample4{

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

System.out.println("? quantifier ....");

System.out.println(Pattern.matches("[amn]?", "a"));//true (a or m or n comes one time)

System.out.println(Pattern.matches("[amn]?", "aaa"));//false (a comes more than one time)

System.out.println(Pattern.matches("[amn]?", "aammmnn"));//false (a m and n comes more than one time)

System.out.println(Pattern.matches("[amn]?", "aazzta"));//false (a comes more than one time)

System.out.println(Pattern.matches("[amn]?", "am"));//false (a or m or n must come one time)

System.out.println("+ quantifier ....");

System.out.println(Pattern.matches("[amn]+", "a"));//true (a or m or n once or more times)

System.out.println(Pattern.matches("[amn]+", "aaa"));//true (a comes more than one time)

System.out.println(Pattern.matches("[amn]+", "aammmnn"));//true (a or m or n comes more than once)

System.out.println(Pattern.matches("[amn]+", "aazzta"));//false (z and t are not matching pattern)

System.out.println("\* quantifier ....");

System.out.println(Pattern.matches("[amn]\*", "ammmna"));//true (a or m or n may come zero or more times)

}}

## Regex Metacharacters/Boundary Matchers

The regular expression metacharacters work as shortcodes.

|  |  |
| --- | --- |
| **Regex** | **Description** |
| . | Any character (may or may not match terminator) |
| ^ | Matches the beginning of the line. |
| \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 |
| | | Find a match for any one of the patterns separated by | as in: cat|dog|fish |
| $ | Finds a match at the end of the string as in: World$ |

## *For e.g. Pattern.matches("^Hello$", "Hello"): return true, Begins and ends with Hello Pattern.matches("^Hello$", "Namaste! Hello"): return false, does not begin with Hello Pattern.matches("^Hello$", "Hello Namaste!"): return false, Does not end with Hello*

## *Java regex word boundary example:*

## *String text = "Mary had a little lamb";*

## *Pattern pattern = Pattern.compile("\\b");*

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

## *while(matcher.find()){*

## *System.out.println("Found match at: " + matcher.start() + " to " + matcher.end());*

## *}*

## *The output of running this example would be:*

## *Found match at: 0 to 0*

## *Found match at: 4 to 4*

## *Found match at: 5 to 5*

## *Found match at: 8 to 8*

## *Found match at: 9 to 9*

## *Found match at: 10 to 10*

## *Found match at: 11 to 11*

## *Found match at: 17 to 17*

## *Found match at: 18 to 18*

## *Found match at: 22 to 22*

## *----------------------*

## *String text = "Mary had a little lamb";*

## *Pattern pattern = Pattern.compile("\\bl");*

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

## *while(matcher.find()){*

## *System.out.println("Found match at: " + matcher.start() + " to " + matcher.end());*

## *}*

## *This example will find all the locations where a word starts with the letter l(lowercase). In fact it will also find the ends of these matches, meaning the last character of the pattern, which is the lowercase l letter.*

## *Java regex non-word boundary example:*

## *The \B boundary matcher matches non-word boundaries. A non-word boundary is a boundary between two characters which are both part of the same word. In other words, the character combination is not word-to-non-word character sequence (which is a word boundary). Here is a simple Java regex non-word boundary matcher example:*

## *String text = "Mary had a little lamb";*

## *Pattern pattern = Pattern.compile("\\B");*

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

## *while(matcher.find()){*

## *System.out.println("Found match at: " + matcher.start() + " to " + matcher.end());*

## *}*

## *This example will give the following output:*

## *Found match at: 1 to 1*

## *Found match at: 2 to 2*

## *Found match at: 3 to 3*

## *Found match at: 6 to 6*

## *Found match at: 7 to 7*

## *Found match at: 12 to 12*

## *Found match at: 13 to 13*

## *Found match at: 14 to 14*

## *Found match at: 15 to 15*

## *Found match at: 16 to 16*

## *Found match at: 19 to 19*

## *Found match at: 20 to 20*

## *Found match at: 21 to 21*

## *Notice how these match indexes corresponds to boundaries between characters within the same word.*

## Regular Expression Metacharacters Example

**import** java.util.regex.\*;

**class** RegexExample5{

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

System.out.println("metacharacters d....");\\d means digit

System.out.println(Pattern.matches("\\d", "abc"));//false (non-digit)

System.out.println(Pattern.matches("\\d", "1"));//true (digit and comes once)

System.out.println(Pattern.matches("\\d", "4443"));//false (digit but comes more than once)

System.out.println(Pattern.matches("\\d", "323abc"));//false (digit and char)

System.out.println("metacharacters D....");\\D means non-digit

System.out.println(Pattern.matches("\\D", "abc"));//false (non-digit but comes more than once)

System.out.println(Pattern.matches("\\D", "1"));//false (digit)

System.out.println(Pattern.matches("\\D", "4443"));//false (digit)

System.out.println(Pattern.matches("\\D", "323abc"));//false (digit and char)

System.out.println(Pattern.matches("\\D", "m"));//true (non-digit and comes once)

System.out.println("metacharacters D with quantifier....");

System.out.println(Pattern.matches("\\D\*", "mak"));//true (non-digit and may come 0 or more times)

}}

## Regular Expression Question 1

/\*Create a regular expression that accepts alphanumeric characters only.

Its length must be six characters long only.\*/

**import** java.util.regex.\*;

**class** RegexExample6{

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

System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun32"));//true

System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "kkvarun32"));//false (more than 6 char)

System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "JA2Uk2"));//true

System.out.println(Pattern.matches("[a-zA-Z0-9]{6}", "arun$2"));//false ($ is not matched)

}}

## Regular Expression Question 2

/\*Create a regular expression that accepts 10 digit numeric characters

 starting with 7, 8 or 9 only.\*/

**import** java.util.regex.\*;

**class** RegexExample7{

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

System.out.println("by character classes and quantifiers ...");

System.out.println(Pattern.matches("[789]{1}[0-9]{9}", "9953038949"));//true

System.out.println(Pattern.matches("[789][0-9]{9}", "9953038949"));//true

System.out.println(Pattern.matches("[789][0-9]{9}", "99530389490"));//false (11 characters)

System.out.println(Pattern.matches("[789][0-9]{9}", "6953038949"));//false (starts from 6)

System.out.println(Pattern.matches("[789][0-9]{9}", "8853038949"));//true

System.out.println("by metacharacters ...");

System.out.println(Pattern.matches("[789]{1}\\d{9}", "8853038949"));//true

System.out.println(Pattern.matches("[789]{1}\\d{9}", "3853038949"));//false (starts from 3)

}}

**More Regex Pattern Example**:

* **Validate numeric value(int or float)** :   
  ^[+-]?\\d\*\\.?\\d\*$   
  ^([+-]?\\d\*\\.?\\d\*)$
* **Validate integer value only** :  
  ^[+-]?\\d\*$
* **Validate date value only** :  
  \\d{4}-\\d{2}-\\d{2}  
  \\d{4}-(0[1-9]|1[012])-(0[1-9]|[12][0-9]|[3][01])

## Java Regex Finder Example

**find()**: Searches the occurrences of of the regular expressions in the text. Mainly used when we are searching for multiple occurrences.

**start() and end()**: Both these methods are generally used along with the find() method. They are used for getting the start and end indexes of a match that is being found using find() method.

#### Lets take an example to find out the multiple occurrences using Matcher methods:

package beginnersbook.com;

import java.util.regex.\*;

class RegexExampleMatcher{

public static void main(String args[]){

String content = "ZZZ AA PP AA QQQ AAA ZZ";

String string = "AA";

Pattern pattern = Pattern.compile(string);

Matcher matcher = pattern.matcher(content);

while(matcher.find()) {

System.out.println("Found at: "+ matcher.start()

+

" - " + matcher.end());

}

}

}

Output:

Found at: 4 - 6

Found at: 10 - 12

Found at: 17 - 19

Now we are familiar with Pattern and Matcher class and the process of matching a regular expression against the text. Lets see what kind of various options we have to define a regular expression:

## Other Example:

**import** java.util.regex.Pattern;

**import** java.util.Scanner;

**import** java.util.regex.Matcher;

**public** **class** RegexExample8{

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

        Scanner sc=**new** Scanner(System.in);

**while** (**true**) {

            System.out.println("Enter regex pattern:");

            Pattern pattern = Pattern.compile(sc.nextLine());

            System.out.println("Enter text:");

            Matcher matcher = pattern.matcher(sc.nextLine());

**boolean** found = **false**;

**while** (matcher.find()) {

                System.out.println("I found the text "+matcher.group()+" starting at index "+

                 matcher.start()+" and ending at index "+matcher.end());

                found = **true**;

            }

**if**(!found){

                System.out.println("No match found.");

            }

        }

    }

}

Output:

Enter regex pattern: java

Enter text: this is java, do you know java

I found the text java starting at index 8 and ending at index 12

I found the text java starting at index 26 and ending at index 30

**Pattern.split() Example**

To split a text into multiple strings based on a delimiter (Here delimiter would be specified using **regex**), we can use Pattern.split() method. This is how it can be done.

*import java.util.regex.\*;*

*class RegexExample2{*

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

*String text = "ThisIsChaitanya.ItISMyWebsite";*

*// Pattern for delimiter*

*String patternString = "is";*

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

*String[] myStrings = pattern.split(text);*

*for(String temp: myStrings){*

*System.out.println(temp);*

*}*

*System.out.println("Number of split strings: "+myStrings.length);*

*}}*

Output:

*Th*

*Chaitanya.It*

*MyWebsite*

*Number of split strings: 4*

The second split String is null in the output.

**Replacement Methods**

Replacement methods are useful methods for replacing text in an input string −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public Matcher appendReplacement(StringBuffer sb, String replacement)**  Implements a non-terminal append-and-replace step. |
| 2 | **public StringBuffer appendTail(StringBuffer sb)**  Implements a terminal append-and-replace step. |
| 3 | **public String replaceAll(String replacement)**  Replaces every subsequence of the input sequence that matches the pattern with the given replacement string. |
| 4 | **public String replaceFirst(String replacement)**  Replaces the first subsequence of the input sequence that matches the pattern with the given replacement string. |
| 5 | **public static String quoteReplacement(String s)**  Returns a literal replacement String for the specified String. This method produces a String that will work as a literal replacement **s** in the appendReplacement method of the Matcher class. |

**The matches and lookingAt Methods**

The matches and lookingAt methods both attempt to match an input sequence against a pattern. The difference, however, is that matches requires the entire input sequence to be matched, while lookingAt does not.

Both methods always start at the beginning of the input string. Here is the example explaining the functionality −

**Example**

*import java.util.regex.Matcher;*

*import java.util.regex.Pattern;*

*public class RegexMatches {*

*private static final String REGEX = "foo";*

*private static final String INPUT = "fooooooooooooooooo";*

*private static Pattern pattern;*

*private static Matcher matcher;*

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

*pattern = Pattern.compile(REGEX);*

*matcher = pattern.matcher(INPUT);*

*System.out.println("Current REGEX is: "+REGEX);*

*System.out.println("Current INPUT is: "+INPUT);*

*System.out.println("lookingAt(): "+matcher.lookingAt());*

*System.out.println("matches(): "+matcher.matches());*

*}*

*}*

This will produce the following result −

**Output**

Current REGEX is: foo

Current INPUT is: fooooooooooooooooo

lookingAt(): true

matches(): false

**The replaceFirst and replaceAll Methods**

The replaceFirst and replaceAll methods replace the text that matches a given regular expression. As their names indicate, replaceFirst replaces the first occurrence, and replaceAll replaces all occurrences.

Here is the example explaining the functionality −

**Example**

*import java.util.regex.Matcher;*

*import java.util.regex.Pattern;*

*public class RegexMatches {*

*private static String REGEX = "dog";*

*private static String INPUT = "The dog says meow. " + "All dogs say meow.";*

*private static String REPLACE = "cat";*

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

*Pattern p = Pattern.compile(REGEX);*

*// get a matcher object*

*Matcher m = p.matcher(INPUT);*

*INPUT = m.replaceAll(REPLACE);*

*System.out.println(INPUT);*

*}*

*}*

This will produce the following result −

**Output**

The cat says meow. All cats say meow.

**The appendReplacement and appendTail Methods**

The Matcher class also provides appendReplacement and appendTail methods for text replacement.

Here is the example explaining the functionality −

**Example**

*import java.util.regex.Matcher;*

*import java.util.regex.Pattern;*

*public class RegexMatches {*

*private static String REGEX = "a\*b";*

*private static String INPUT = "aabfooaabfooabfoob";*

*private static String REPLACE = "-";*

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

*Pattern p = Pattern.compile(REGEX);*

*// get a matcher object*

*Matcher m = p.matcher(INPUT);*

*StringBuffer sb = new StringBuffer();*

*while(m.find()) {*

*m.appendReplacement(sb, REPLACE);*

*}*

*m.appendTail(sb);*

*System.out.println(sb.toString());*

*}*

*}*

This will produce the following result −

**Output**

-foo-foo-foo-

**Capturing Groups**

Capturing groups are a way to treat multiple characters as a single unit. They are created by placing the characters to be grouped inside a set of parentheses. For example, the regular expression (dog) creates a single group containing the letters "d", "o", and "g".

Capturing groups are numbered by counting their opening parentheses from the left to the right. In the expression ((A)(B(C))), for example, there are four such groups −

* ((A)(B(C)))
* (A)
* (B(C))
* (C)

To find out how many groups are present in the expression, call the groupCount method on a matcher object. The groupCount method returns an **int** showing the number of capturing groups present in the matcher's pattern.

There is also a special group, group 0, which always represents the entire expression. This group is not included in the total reported by groupCount.

**Example**

Following example illustrates how to find a digit string from the given alphanumeric string −

*import java.util.regex.Matcher;*

*import java.util.regex.Pattern;*

*public class RegexMatches {*

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

*// String to be scanned to find the pattern.*

*String line = "This order was placed for QT3000! OK?";*

*String pattern = "(.\*)(\\d+)(.\*)";*

*// Create a Pattern object*

*Pattern r = Pattern.compile(pattern);*

*// Now create matcher object.*

*Matcher m = r.matcher(line);*

*if (m.find( )) {*

*System.out.println("Found value: " + m.group(0) );*

*System.out.println("Found value: " + m.group(1) );*

*System.out.println("Found value: " + m.group(2) );*

*}else {*

*System.out.println("NO MATCH");*

*}*

*}*

*}*

This will produce the following result −

**Output**

Found value: This order was placed for QT3000! OK?

Found value: This order was placed for QT300

Found value: 0

PatternSyntaxException Class Methods

A PatternSyntaxException is an unchecked exception that indicates a syntax error in a regular expression pattern. The PatternSyntaxException class provides the following methods to help you determine what went wrong −

|  |  |
| --- | --- |
| **Sr.No.** | **Method & Description** |
| 1 | **public String getDescription()**  Retrieves the description of the error. |
| 2 | **public int getIndex()**  Retrieves the error index. |
| 3 | **public String getPattern()**  Retrieves the erroneous regular expression pattern. |
| 4 | **public String getMessage()**  Returns a multi-line string containing the description of the syntax error and its index, the erroneous regular expression pattern, and a visual indication of the error index within the pattern. |