**Regular expressions** are used for defining String patterns that can be used for searching, manipulating and editing a text. These expressions are also known as **Regex** (short form of Regular expressions).

#### Lets take an example to understand it better:

In the below example, the regular expression .\*book.\* is used for searching the occurrence of string “book” in the text.

import java.util.regex.\*;

class RegexExample1{

public static void main(String args[]){

String content = "This is Chaitanya " +

"from Beginnersbook.com.";

String pattern = ".\*book.\*";

boolean isMatch = Pattern.matches(pattern, content);

System.out.println("The text contains 'book'? " + isMatch);

}

}

Output:

The text contains 'book'? true

In this tutorial we will learn how to define patterns and how to use them. The java.util.regex API (the package which we need to import while dealing with Regex) has two main classes:

1) java.util.regex.Pattern – Used for defining patterns  
2) java.util.regex.Matcher – Used for performing match operations on text using patterns

## java.util.regex.Pattern class:

#### 1) Pattern.matches()

We have already seen the usage of this method in the above example where we performed the search for string “book” in a given text. This is one of simplest and easiest way of searching a String in a text using Regex.

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

String patternString = ".\*tutorial.\*";

boolean isMatch = Pattern.matches(patternString, content);

System.out.println("The text contains 'tutorial'? " + isMatch);

As you can see we have used matches() method of Pattern class to search the pattern in the given text. The pattern .\*tutorial.\* allows zero or more characters at the beginning and end of the String “tutorial” (the expression .\* is used for zero and more characters).

**Limitations**: This way we can search a single occurrence of a pattern in a text. For matching multiple occurrences you should use the Pattern.compile() method (discussed in the next section).

#### 2) Pattern.compile()

In the above example we searched a string “tutorial” in the text, that is a case sensitive search, however if you want to do a CASE INSENSITIVE search or want to do search multiple occurrences then you may need to first compile the pattern using Pattern.compile() before searching it in text. This is how this method can be used for this case.

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

String patternString = ".\*tuToRiAl.";

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

Here we have used a flag Pattern.CASE\_INSENSITIVE for case insensitive search, there are several other flags that can be used for different-2 purposes. To read more about such flags [refer this document](https://docs.oracle.com/javase/tutorial/essential/regex/pattern.html).

**Now what**: We have obtained a Pattern instance but how to match it? For that we would be needing a Matcher instance, which we can get using Pattern.matcher() method. Lets discuss it.

#### 3) Pattern.matcher() method

In the above section we learnt how to get a Pattern instance using compile() method. Here we will learn How to get Matcher instance from Pattern instance by using matcher() method.

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

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

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

Matcher matcher = pattern.matcher(content);

boolean isMatched = matcher.matches();

System.out.println("Is it a Match?" + isMatched);

Output:

Is it a Match?true

#### 4) Pattern.split()

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.

## java.util.regex.Matcher Class

We already discussed little bit about Matcher class above. Lets recall few things:

#### Creating a Matcher instance

String content = "Some text";

String patternString = ".\*somestring.\*";

Pattern pattern = Pattern.compile(patternString);

Matcher matcher = pattern.matcher(content);

#### Main methods

**matches()**: It matches the regular expression against the whole text passed to the Pattern.matcher() method while creating Matcher instance.

...

Matcher matcher = pattern.matcher(content);

boolean isMatch = matcher.matches();

**lookingAt()**: Similar to matches() method except that it matches the regular expression only against the beginning of the text, while matches() search in the whole text.

**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:

#### 1) String Literals

Lets say you just want to search a particular string in the text for e.g. “abc” then we can simply write the code like this: Here text and regex both are same.  
Pattern.matches("abc", "abc")

#### 2) Character Classes

A character class matches a single character in the input text against multiple allowed characters in the character class. For example [Cc]haitanya would match all the occurrences of String “chaitanya” with either lower case or upper case C”. Few more examples:  
Pattern.matches("[pqr]", "abcd"); It would give false as no p,q or r in the text  
Pattern.matches("[pqr]", "r"); Return true as r is found  
Pattern.matches("[pqr]", "pq"); Return false as any one of them can be in text not both.

Here is the complete list of various character classes constructs:  
[abc]: It would match with text if the text is having either one of them(a,b or c) and only once.  
[^abc]:  Any single character except a, b, or c (^ denote 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]]:  Any one of them (d, e, or f)  
[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)

#### Predefined Character Classes – Metacharacters

These are like short codes which you can use while writing regex.

Construct Description

. -> 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]

For e.g.  
Pattern.matches("\\d", "1"); would return true  
Pattern.matches("\\D", "z"); return true  
Pattern.matches(".p", "qp"); return true, dot(.) represent any character

#### Boundary Matchers

^ Matches the beginning of a line.

$ Matches then end of a line.

\b Matches a word boundary.

\B Matches a non-word boundary.

\A Matches the beginning of the input text.

\G Matches the end of the previous match

\Z Matches the end of the input text except the final terminator if any.

\z Matches the end of the input text.

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

#### Quantifiers

Greedy Reluctant Possessive Matches

X? X?? X?+ Matches X once, or not at all (0 or 1 time).

X\* X\*? X\*+ Matches X zero or more times.

X+ X+? X++ Matches X one or more times.

X{n} X{n}? X{n}+ Matches X exactly n times.

X{n,} X{n,}? X{n,}+ Matches X at least n times.

X{n, m) X{n, m)? X{n, m)+ Matches X at least n time, but at most m times.

#### Few examples

import java.util.regex.\*;

class RegexExample{

public static void main(String args[]){

// It would return true if string matches exactly "tom"

System.out.println(

Pattern.matches("tom", "Tom")); //False

/\* returns true if the string matches exactly

\* "tom" or "Tom"

\*/

System.out.println(

Pattern.matches("[Tt]om", "Tom")); //True

System.out.println(

Pattern.matches("[Tt]om", "Tom")); //True

/\* Returns true if the string matches exactly "tim"

\* or "Tim" or "jin" or "Jin"

\*/

System.out.println(

Pattern.matches("[tT]im|[jJ]in", "Tim"));//True

System.out.println(

Pattern.matches("[tT]im|[jJ]in", "jin"));//True

/\* returns true if the string contains "abc" at

\* any place

\*/

System.out.println(

Pattern.matches(".\*abc.\*", "deabcpq"));//True

/\* returns true if the string does not have a

\* number at the beginning

\*/

System.out.println(

Pattern.matches("^[^\\d].\*", "123abc")); //False

System.out.println(

Pattern.matches("^[^\\d].\*", "abc123")); //True

// returns true if the string contains of three letters

System.out.println(

Pattern.matches("[a-zA-Z][a-zA-Z][a-zA-Z]", "aPz"));//True

System.out.println(

Pattern.matches("[a-zA-Z][a-zA-Z][a-zA-Z]", "aAA"));//True

System.out.println(

Pattern.matches("[a-zA-Z][a-zA-Z][a-zA-Z]", "apZx"));//False

// returns true if the string contains 0 or more non-digits

System.out.println(

Pattern.matches("\\D\*", "abcde")); //True

System.out.println(

Pattern.matches("\\D\*", "abcde123")); //False

/\* Boundary Matchers example

\* ^ denotes start of the line

\* $ denotes end of the line

\*/

System.out.println(

Pattern.matches("^This$", "This is Chaitanya")); //False

System.out.println(

Pattern.matches("^This$", "This")); //True

System.out.println(

Pattern.matches("^This$", "Is This Chaitanya")); //False

}

}