|  |
| --- |
| Regular Expression |

A Regular Expression or Regex is a special sequence of characters that help in matching or finding other strings or set of strings, using a specialized syntax. Regex are used in searching, editing and manipulating the text.

The Regex API (java.util.regex) provides the necessary classes and interfaces for working with regular expressions. String class uses this API to support regex in its following methods: matches(), split(), replaceFirst(), and replaceAll(). Let us concentrate on the methods *matches()* and *split()* for now.

* **matches()**: This method takes a regex as an argument and matches a given string with the specified regex pattern. This method returns a Boolean value for the comparison done.
  1. *//Creating a Pattern through Regular Expression*
  2. String regex = "Hello";
  3. *//String to be compared with RegEx*
  4. String input = "Hello";
  6. *//Pattern Matching done using matches() method*
  7. System.out.println(input.matches(regex));
  8. *//Pattern Matching done using Pattern.matches() method*
  9. System.out.println(Pattern.matches(regex, input));
  10. */\*\**
  11. *\* Output: true, true*
  12. *\*/*

It is interesting to note that,*input.matches(regex)* yields the same result as *Pattern.matches(regex, input)*.(Note: Pattern is one of the classes provided by the Regex API)

* **split()**: This method splits a given string into an array of string on the basis of regular expression mentioned in the argument of the method.
  1. *//Sample String*
  2. String chainedString = "Lets-break-this-chain-and-be-free-!";
  3. *//Creating Regex for splitting the String*
  4. String regex = "-";
  5. *//Splitting the String using split() method*
  6. *//split()divides the String and stores them in a Array of String*
  7. String[] freeString = chainedString.split(regex);
  8. */\*\**
  9. *\* freeString = ("Lets", "break", "this", "chain", "and", "be", "free", "!")*
  10. *\*/*

Now let's dive deeper and understand how to write Regular Expressions.

While working with strings, we must have wondered if there was an easier way to validate them. What if we could specify a required pattern directly instead of checking things character by character? What if we could provide a general pattern to search for instead of repeatedly looking for specific variations? And to make this possible, Java supports Regular Expressions.

Now the question arises: "How can we write such 'generalized' regular expressions?"

And before we go searching for the answers, let us see what a general regular expression looks like:

1. *//Regular Expression for Name Validation:*
2. String nameRegex = "([A-Za-z ]+)";
3. *//Regular Expression for Email Id Validation:*
4. String emailIdRegex = "([A-Za-z0-9-\_]+)[@]([a-z]+)[.](com|in)";

To write a generalized regular expression, we require the help of the following:

* Literals: They are the literal characters that have a literal meaning.
  1. String literalRegex = "Hello"; *//The string must be exactly "Hello" to match this Regex pattern*

 Metacharacters.

Let's take a look at the different metacharacters.

| **Meta Characters** | | |
| --- | --- | --- |
| **Meta Character** | **Description** | **Example** |
| **[]** | **Bracket expressions create a character class to match a single character  contained within the brackets. '\_' can be used to specify a range.** | **[xyz] matches 'x', 'y' or 'z'. [a-z] matches any**  **letter from 'a' to 'z'.** |
| **.** | **Matches any single character, except a newline. Inside a bracket expression, it becomes a literal dot.** | **b.t matches "bat", "bRt", "b8t", etc.** |
| **[^ ]** | **Matches a single character that is not within the brackets.** | **[^xyz] matches 'a', '6', etc.** |
| **I** | **"or" expression to match alternatives.** | **bat|cat matches "bat" or "cat"** |
| **()** | **Groups expressions to form sub-expressions. Also used to capture groups.** | **Ma(nn|tt)er matches "Matter" or "Manner"** |
| **\n** | **Matches the nth captured sub-expression group. Groups are numbered from left to right.** | **([a-z])\1 matches "dd", "hh", etc.** |

**Note**:

* If meta characters are needed to be used as literals, they have to be escaped with a double backslash.
* To refer to captured groups in the replacement methods, $<group no.> is used. E.g. str.replaceAll(regex, "$1");

Next, let us look at quantifiers.

Quantifiers:

Let's take a look at the different quantifiers.

| **Quantifiers** | | |
| --- | --- | --- |
| **Meta Character** | **Description** | **Example** |
| **?** | **Matches the preceding element zero or one time.** | **Ba?it matches "Bait" and "Bit"** |
| **\*** | **Matches the preceding element zero or more times.** | **10\*1 matches "11", "1001", etc.** |
| **+** | **Matches the preceding element one or more times.** | **10+1 matches "101", "1001", etc.** |
| **{m}** | **Matches the preceding element exactly m times.** | **10{4}1 matches "100001"** |
| **{m,}** | **Matches the preceding element m or more times.** | **10{3,}1 matches "10001", "100001", "1000000001", etc.** |
| **{m,n}** | **Matches the preceding element minimum m and   maximum n times.** | **xy{2,3}z matches "xyyz" and "xyyyz"** |

Let us look at some predefined character sets next.

Predefined Character Classes:

Now that we have seen different metacharacters and quantifiers, let's see some predefined classes used in Regex. These "predefined classes" can be used as alternatives to standard Regex written using only meta characters and literals.

| **ArrayListIterator methods** | | |
| --- | --- | --- |
| **Meta Character** | **Description** | **Example** |
| **\w** | **Alphanumeric characters and the underscore** | **[A-Za-z0-9\_]** |
| **\W** | **Non-word characters** | **[^A-Za-z0-9\_]** |
| **\d** | **Digits** | **[0-9]** |
| **\D** | **Non-digits** | **[^0-9]** |
| **\s** | **Whitespace characters** | **[\t\n\f\r)** |
| **\S** | **Non-whitespace characters** | **[^\t\n\f\r]** |

**Note**:

* "\w" is a meta character. But in Java, '\' is an escape character. Hence, we have to escape it using another backslash. E.g. "[\\w]+"

Now, let us see how you can write a regular expression.

Approaching Regular Expression:

Let's now see how to approach a pattern matching problem using the Regular Expression method. But before we proceed, it must be noted and understood that there can be multiple solutions to a problem.

Let's consider a situation where we have to validate a Courier Consignment Tracking Number. The required format/pattern for the tracking number is given as:

1. Format of Courier Consignment Tracking Number:
2. <<2\_Uppercase\_Characters>>-<<9\_digits>>:<<2\_Uppercase\_Characters>>
3. Example: "EK-860619645:IN"

Let us observe all the character sets the string is containing. This pattern has only Uppercase Alphabets, Digits and Two Special Characters '-' and ':'. Next, let us find the frequency of occurrences for the character classes. With that, let's break the pattern into smaller sub-patterns:

* The first sub-pattern is having two uppercase characters followed by "-" character. So, the Regex for this sub-pattern can be written as:
  1. String firstSubRegex = "[A-Z]{2}[-]";

* The second sub-pattern has 9 digits followed by ":" character. So, the Regex for this sub-pattern can be written as:
  1. *//First Alternative:*
  2. String secondSubRegex = "[0-9]{9}[:]";
  3. *//Another Alternative:*
  4. String secondSubRegex = "[\\d]{9}[:]";

* The last sub-pattern is having only two uppercase characters again. So, the Regex for this sub-pattern can be written as:
  1. String thirdSubRegex = "[A-Z]{9}"

Hence, the overall matching pattern can be written as:

1. *//Concatenation of all three Sub-Regex:*
2. String finalRegexPatternConcat = firstSubRegex + secondSubRegex + thirdSubRegex;
3. *//Alternatively the single Regex:*
4. String finalRegexPatternComplete = "([A-Z]{2}[-])([\\d]{9}[:])([A-Z]{2})";

Now, let  us look at some more examples on Regular Expressions.

Now let us conclude Regular Expressions by looking into following situational examples:

1. Requirement: To search the pattern "App" in the given string "Application".
   1. String searchStr = "Application";
   2. String regexStr = "App.\*";
   3. System.out.println(searchStr.matches(regexStr)); *//Output: true*

1. Requirement: To search the pattern having two characters in between A and l in the given string "A%(lication".
   1. String searchStr = "A%(lication";
   2. String regexStr = "A..lication";
   3. System.out.println(searchStr.matches(regexStr)); *//Output: true*

1. Requirement: To search for a digit between M and t in the given string "M4thematics".
   1. String searchStr = "M4thematics";
   2. String regexStr = "M\\dt.\*";
   3. System.out.println(searchStr.matches(regexStr)); *//Output: true*

1. Requirement: To search for a number between 4 and 8 in between X and Y in the given string.
   1. String searchStr = "X9Y";
   2. String regexStr = "X[4-8]Y";
   3. System.out.println(searchStr.matches(regexStr)); *//Output: false*

1. Requirement: To search for the pattern "Hell" or "Fell" in the given string "Fellow".
   1. String searchStr = "Fellow";
   2. String regexStr = "(Hell|Fell).\*";
   3. System.out.println(searchStr.matches(regexStr)); *//Output: true*

1. Requirement: To check for the space after "Air" in the given string "Air line".
   1. String searchStr = "Air line";
   2. String regexStr = "Air\\s.\*";
   3. System.out.println(searchStr.matches(regexStr)); *//Output: true*

1. Requirement: To check if a number is found 0 or n times after X in the given string.
   1. String searchStr = "X4756Y";
   2. String regexStr = "X\\d\*Y";
   3. System.out.println(searchStr.matches(regexStr)); *//Output: true*

1. Requirement: To check if a number is found 1 or n times after M in the given string.
   1. String searchStr = "M4N";
   2. String regexStr = "M\\d+N";
   3. System.out.println(searchStr.matches(regexStr)); *//Output: true*

1. Requirement: To check if a number is found 0 or 1 times after A in the given string.
   1. String searchStr = "M2N";
   2. String regexStr = "M\\d?N";
   3. System.out.println(searchStr.matches(regexStr)); *//Output: true*

1. Requirement: To check if 3 digits are present after A in the given string.
   1. String searchStr = "M42N";
   2. String regexStr = "M\\d{3}N";
   3. System.out.println(searchStr.matches(regexStr)); *//Output: false*

Now, let's look further into grouping of Regular Expressions.

 Grouping of a Regular Expression is done to divide a complete pattern into smaller groups of the pattern. This is done by using the parenthesis "()" brackets. Let us look at the various situations where grouping is necessary:

1. Grouping is necessary to group a combination of letters or words together:
   1. *//Regex to search between two alternatives: "Scanner" or "Scammer":*
   2. String regex = "Sca(nn|mm)er";
   3. String demoStr = "Scammer";
   4. System.out.println(demoStr.matches(regex));
   5. *//Regex to search for a particular word in a String*
   6. String regex = ".\*(dog).\*";
   7. String demoStr = "Bowser was my dog."
   8. System.out.println(demoStr.matches(regex));
2. Grouping is necessary to divide a bigger pattern into smaller patterns. Let us take a previous example to observe this better. We had the format for Courier Consignment Tracking Number as "*<<2\_Uppercase\_Characters>>-<<9\_digits>>:<<2\_Uppercase\_Characters>>*". And this pattern could be divided into the following three parts:
   1. Regex Sub-Pattern 1: The first sub-pattern is having two uppercase characters followed by "-" character - "[A-Z]{2}[-]"
   2. Regex Sub-Pattern 2: The second sub-pattern has 9 digits followed by ":" character - "[\\d]{9}[:]"
   3. Regex Sub-Pattern 3: The last sub-pattern is having only two uppercase characters again  - "[A-Z]{2}"
      1. *//Grouping of Regex done according to Sub-patterns using "()":*
      2. String regex = "([A-Z]{2}[-])([\\d]{9}[:])([A-Z]{2})";

1. Grouping is necessary for validating repeating/non-repeating sequences. In order to do so, we also use Back Referencing along with Grouping:
   1. *//Regex to search for repeating sequence in the String:*
   2. String passCodeRegex = ".\*(\\d)\\1+.\*";
   3. String passCodeStr = "2455254";
   4. System.out.println(passCodeStr.matches(passCodeRegex));

Here, "\\1" after "(\\d)" in the regular expression is called a backreference. It helps in checking the group mentioned in the regex is repeating or not.

* 1. *//Regex having three groups being Back-Referenced*
  2. *//The three groups must repeat after the ":" symbol in the String to match the pattern.*
  3. String repeatRegex = "([A-Za-z0-9]+) (\\d+) ([A-Z]+) [:] \\1 \\2 \\3";
  4. String repeatStr = "Tom123 9090 JERRY : Tom123 9090 JERRY";

In the above example, "\\1" refers to the first group "([A-Za-z0-9])", "\\2" refers to "(\\d+)" and "\\3" refers to "([A-Z]+)". And so, the regex expects the string to have all the three groups repeat itself whenever the back referencing is done.

# **Regular Expression - Exercise 1**

Problem Statement:

Write a program to validate the given web address is a valid one or not according to the following guidelines:

* It should start with either http or https
* It should be followed by "://"
* It may have "www."
* The web address should be alphanumeric
* After a "." the domain name should be one among "com", "org", "net"

Create a class, WebAddressValidator, as shown below, to implement the logic.

**WebAddressValidator:**



**Method Description:**

**isValidWebAddress(String webAddress):** This method accepts the webAddress entered by the user and validates it.

To test the above functionality, make a **Tester** class.

**Sample:**

**Input (Valid):** "http://www.microsoft.com"

**Output:**"You have entered a valid web address"

**Input (Invalid):** "http.www.abc.au"

**Output:** "You have entered an invalid web address"

Note: Check the project using SonarLint to maintain the coding standards. Ignore the violations which occur due to "System.out" statements.

.